January 2015

Optimizing Virtual Machine I/O Performance in Virtualized Cloud by Differenciated-frequency Scheduling and Functionality Offloading

Cong Xu
Purdue University

Graduate School Form 30
Updated 1/15/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By CONG XU

Entitled
Optimizing Virtual Machine I/O Performance in Virtualized Cloud by Differentiated-frequency Scheduling and Functionality Offloading

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Dr. Dongyan Xu, Chair	Dr. Ramana Kompella
Dr. Patrick Eugster	
Dr. Yung-Hsiang Lu	
Dr. Sonia Fahmy	

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Dr. Dongyan Xu

Approved by: Dr. Sunil Prabhakar/Dr. William J Gorman 12/4/2015

Head of the Departmental Graduate Program Date

OPTIMIZING VIRTUAL MACHINE I/O PERFORMANCE IN VIRTUALIZED CLOUD BY DIFFERENCIATED-FREQUENCY SCHEDULING AND FUNCTIONALITY OFFLOADING

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Cong Xu

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2015

Purdue University

West Lafayette, Indiana

To my wife and parents.

ACKNOWLEDGMENTS

First of all, I would like to express my deepest gratitude to my advisor, Professor Dongyan Xu. Thank him for the invaluable guidance, caring, encouragement, and support for the last five years. He is the most influential and most important person who introduced me into computer science research and taught me how to become a good researcher in all aspects. He always encourage me to explore new research areas and give me the freedom to propose new ideas. From him, I learned how to initiate an idea, how to differentiate it from existing research efforts, how to evaluate the merits of other researchers' work, how to structure a paper, and how to deliver my research ideas to the audience in the most efficient way.

Secondly, I would like to express my sincere thanks to Professor Ramana Kompella who is my co-advisor. We have been working closely since my second semester at Purdue. He greatly broaden my view on the computer systems with his expertise on networking and distributed system. I enjoy the time discussing research ideas with him. He can always give innovative inspirations and provide constructive suggestions.

I would like to extend my thanks to Professor Patrick Eugster, Professor Yung-Hsiang Lu, and Professor Sonia Fahmy for serving in my final examining committee and prelim committee. Their suggestions on my dissertation are very valuable and important to me. I would also like to thank Dr. Antonio Lain of HP Labs. I really enjoyed the time of being an intern at Palo Alto in the summer of 2012.

Finally, I would like to thank my parents, Qingtai Xu and Shuhua Tian, for their understanding, sacrifice, and support for me. I would also like to express my deepest love to my wife, Xuan Yang. I cannot imagine I could finish this dissertation without her standing behind me. During the past 5 years, she always gave me the strongest support no matter what difficulties I met.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

ACK	Acknowledgment
CDF	Cumulative Density Function
CPU	Central Processing Unit
EMR	Elastic Map/Reduce
HDFS	Hadoop Distributed File System
HVE	VMwares Hadoop Virtualization Extention
I/O	Input/Output
IaaS	Infrastructure as a Service
IRQ	Interrupt Request
LAN	Local Area Network
LSVM	Latency Sensitive VM
NLSVM	Non-latency-sensitive VM
MPI	Message Passing Interface
MSS	Maximum Segment Size
NIC	Network Interface Card
OS	Operating System
PaaS	Platform as a Service
PCPU	Physical CPU
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
RTT	Round-Trip Time
ROCE	RDMA over converged Ethernet
SaaS	Software as a Service
TCP	Transmission Control Protocol

TOE	TCP Offload Engine
UDP	User Datagram Protocol
WAN	Wide Area Network
VCPU	Virtual CPU
VM	Virtual Machine
VMM	Virtual Machine Monitor

ABSTRACT

Cong, Xu PhD, Purdue University, December 2015. Optimizing Virtual Machine I/O Performance in Virtualized Cloud by Differenciated-frequency Scheduling and Functionality Offloading. Major Professors: Dongyan Xu and Ramana Rao Kompella.

Many enterprises are increasingly moving their applications to private cloud environments or public cloud platforms. A key technology driving cloud computing is virtualization which can serve multiple VMs in one physical machine hence providing better management flexibility and significant savings in operational costs. However, one important consequence of virtualized hosts in the cloud is the negative impact it has on the I/O performance of the applications running in the VMs.

In this dissertation, we demonstrate that the negative impact of virtualized hosts is mainly caused by two reasons. One is VM consolidation, the other one is virtualization device overhead. First, to alleviate the negative impact of VM consolidation on I/O performance, we introduce two solutions *vSlicer* and *vTurbo*. *vSlicer* enables more timely processing of I/O events by latency sensitive VMs (LSVMs), without violating the CPU share fairness among all CPU sharing VMs. *vTurbo* is a system that accelerates I/O processing for VMs by offloading I/O processing to a designated core, hence significantly improving the VMs network and disk I/O throughput. Second, we show that data movement in the cloud may incur tremendous overhead on different protection layers. Especially, when we directly move bigdata systems such as Hadoop to a virtualized cloud, we observe that device virtualization overhead affects I/O performance of the Hadoop distributed file system (HDFS). My developed work *vRead*, which enables ”direct” read to the disk image of HDFS datanode VM at the hypervisor layer, can avoid most of the virtulization associated overheads and hence improve the I/O performance of applications running in the VMs.

1 INTRODUCTION

Cloud computing is arguably one of the most transformative trends in recent times. Many enterprises and businesses are increasingly migrating their applications to public cloud offerings such as Amazon EC2 [1] and Microsoft Azure [2]. By purchasing or leasing cloud servers with a pay-as-you-go charging model, enterprises benefit from significant cost savings in running their applications, both in terms of capital expenditure (e.g., reduced server costs) as well as operational expenditure (e.g., management staff). On the other hand, cloud providers generate revenue by achieving good performance for their "tenants" while maintaining reasonable cost of operation.

1.1 Thesis Statement

One of the key factors influencing the cost of cloud platforms is *server consolidation*—the ability to host multiple virtual machines (VM) in the same physical server. If the cloud providers can increase the level of server consolidation, i.e., pack more VMs in each physical machine, they can generate more revenue from their infrastructure investment and possibly pass cost savings on to their customers. Two main resources that typically dictate the level of server consolidation, memory and CPU. Memory is strictly partitioned across VMs, although there are techniques (e.g., memory ballooning [3]) for dynamically adjusting the amount of memory available to each VM. CPU can also be strictly partitioned across VMs, with the trend of ever increasing number of cores per physical host. However, given that each core is quite powerful, another scaling factor comes by allocating multiple VMs per core. With the current trend of increasing the core count in multi-core systems, there is a possibility of allocating one core per-VM. However this is not likely to happen in the foreseeable future as shown in the current practice (*e.g.*, Amazon EC2 platform), since we cannot completely eliminate the need for packing multiple VMs in one core , which may be

needed for accommodating surge of VM count and for saving power (by turning off some cores).

In practice, CPU sharing among VMs can be quite complicated. Each VM is typically assigned one or more virtual CPUs (vCPUs) which are scheduled by the hypervisor on to physical CPUs (pCPUs) ensuring proportional fairness. The number of vCPUs is usually larger than the number of pCPUs, which means that, even if a vCPU is ready for execution, it may not find a free pCPU immediately and thus needs to wait for its turn, causing *CPU access latency*. If a VM is running I/O-intensive applications, this latency can have a significant negative impact on application performance [4–8]. To explain this more closely, let us look at I/O processing in modern OSes today. There are two basic stages involved typically. (1) Device interrupts are processed *synchronously* in an IRQ context in the kernel and the data (e.g., network data, disk reads) is buffered in kernel buffers; (2) The application eventually copies the data from kernel buffer to its user-level buffer in an *asynchronous* fashion whenever it gets scheduled by the process scheduler. If the OS were running directly on a physical machine, or if there were a dedicated CPU for a given VM, both procedures can be done almost instantaneously. However, for a VM that shares CPU with other VMs, these two I/O processing procedures may be significantly delayed because the VM may not be running when the I/O event (e.g., network packet arrival) occurs. Eventually, the I/O processing delay negatively impact on applications' I/O performance, resulting in high latency and low throughput.

Besides the CPU access delay caused by VM consolidation, I/O intensive applications like Hadoop/HDFS also suffer from another overhead called device virtualization overhead, because there is additional layer *Hypervisor* between OS and hardware compared with traditional host machine. In virtualized hosts, we observed that running Hadoop inside VMs can lead to sub-optimal performance due to virtualization and data movement overheads. Specifically, the performance of Hadoop inside VMs is heavily dependent on the I/O efficiency of the Hadoop distributed file system (HDFS) [9], because all consumed data by big data applications is first loaded from HDFS. In general, when the client application requests the HDFS datanode to read a file, it reads that file from the local disk and sends its content

back to the client over a TCP socket. Depending on the location of the datanode in relation to the client, this performance can vary drastically. For instance, if there is a co-located datanode, standard Hadoop implementations prefer a *local read* from the co-located datanode over other replicas elsewhere. While the local read is efficient when Hadoop is run in non-virtualized environments, its performance can suffer when the client and datanode are co-located on the same physical host but in different VMs (recommended deployments by Docker [10] and VMware's HVE [11, 12]), due to device virtualization overheads and data movement through protection boundaries (hypervisor, OS, application). *Remote reads* are even slower because of the additional network data transfer overheads.

This dissertation addresses the challenge of alleviating the negative impacts of virtualization imposed on I/O intensive applications in the cloud. First, it examines the VM scheduling induced latency on network packet processing and proposes a solution to reduce network latency achieved by applications running in VMs. Second, it analyses the negative impact on IRQ processing caused by VM consolidation and propose a solution to boost IRQ processing hence improving network and disk I/O throughput. Third, it explores overheads induced by the virtualization layer in data copies of HDFS and propose an "direct" read to the disk image of datanode VM at hypervisor layer.

The thesis of this dissertation is as follows: *Virtualized hosts and VM consolidation negatively impacts the I/O performance of virtual machines. Differentiated VM scheduling, VM IRQ boosting and Efficient Data Access for Hadoop in Virtualized Clouds can significantly reduce I/O latency and improve the I/O throughput of virtual machines.*

1.2 Contributions

My research aims to improve the I/O performance and reduce the device virtualization overhead for the VMs in virtualized cloud. At the same time, we guarantee that the resource fair-share among all VMs running in the same host is still maintained. All proposed methods are transparent to the user-level applications. We implemented all these methods

in the popular hypervisors (Xen or KVM) and eventually constitute an I/O optimization framework for VMs.

The contributions of this dissertation can be summarized as follows:

1. We propose a new class of CPU-sharing VMs called LSVMs to mitigate the impact of VM consolidation on I/O processing latency in VM-hosting clouds. LSVMs achieve much better performance for I/O-bound applications while maintaining the same cost benefit and CPU-share fairness across all sharing VMs.

2. We develop a simple, effective technique called vSlicer to realize LSVMs. Based on the idea of differentiated-frequency microslicing, vSlicer enhances the CPU scheduler of the hypervisor by scheduling LSVMs with smaller microslices but with higher frequency while scheduling NLSVMs with regular (larger) slices, giving I/O-bound VMs more timely access to the CPU for I/O processing without penalizing the NLSVMs' CPU shares.

3. We have implemented a prototype of vSlicer in the Xen hypervisor and conducted extensive evaluation with both micro-benchmarks and application benchmarks. Our micro-benchmark evaluation shows that vSlicer significantly reduces network packet round-trip times (RTTs) and packet jitter (by 70% compared to the vanilla Xen scheduler). Our evaluation with application benchmarks shows substantial improvement in application-specific performance metrics. For example, in our experiments, vSlicer *doubles* both the connection rate and request processing throughput of an Apache web server; reduces a VoIP server's upstream jitter *by 62%*; and shortens the execution times of Intel MPI benchmark programs *by half or more*.

4. We propose a new class of high-frequency scheduling CPU core named turbo core and a new class of co-vCPU called turbo vCPU. The turbo vCPU pinned on turbo core(s) is used for timely processing of the I/O IRQs thus accelerating I/O processing for VMs in the same physical host.

5. We develop a simple but effective VM scheduling policy named vTurbo giving general CPU cores and turbo core magnitudes different time-slice. The very small CPU time-slice on turbo cores grants VM low scheduling delay and low I/O IRQ processing latency.

6. We have implemented a prototype of vTurbo based on Xen. Various evaluations prove the effectiveness of vTurbo. Our micro-benchmark results show that vTurbo can significantly improve the TCP throughput (by up to 3×), UDP throughput (by up to 4×), and disk write throughput (by up to 2×). Our evaluation with application-level benchmarks shows that vTurbo improves application-specific performance as well. For example, Olio's throughput is increased by 38.7%. NFS' throughput is improved by up to 2×.

7. We propose a new file operation interface for HDFS client VMs which allows Hadoop applications to read data from HDFS more efficiently.

8. We develop the vRead system, which provides I/O shortcuts at the hypervisor level via components in the guest and in the hypervisor. vRead works for both *virtual local read* (read from co-located datanode VMs) and *remote read*.

9. We present evaluation results from a vRead prototype implemented in KVM. Our microbenchmark results show that vRead achieves higher read throughput, lower latency, and less CPU consumption compared to standard HDFS running on VMs. For example, Hadoop's throughput can be improved by up to 60% for read and 150% for re-read. Results from a number of Hadoop benchmarks also show significant application-level performance improvements with vRead.

1.2.1 Overview of the VM I/O Optimization Framework

Figure 1.1 illustrates the architectural overview of our optimization framework. Each key component in this framework represents one concrete work during my PhD study. Among these works, vSlicer is a differentiated-frequency VM scheduler used to reduce the

CPU access delay. vTurbo offloads the IRQ processing to a turbo-sliced designated core. vPipe aims to shortcut the data movement between virtual devices within the same VM. While, vRead targets the shortcutting of data copies between different VMs. vHaul focuses on tuning the VM migration sequence for multi-tier applications. In my dissertation, I will mainly demonstrate 3 of them, vSlicer, vTurbo and vRead.

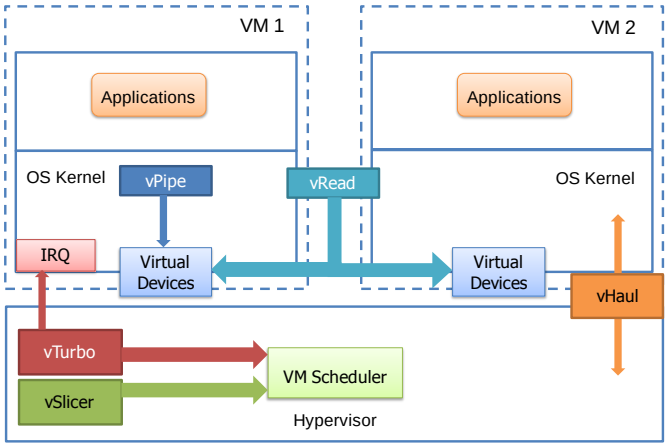


Figure 1.1.: Five key components of our I/O optimization framework.

1.2.2 Low Latency VM Scheduler

As more VMs share the same core/CPU, the CPU access latency experienced by each VM increases substantially, which translates into longer I/O processing latency perceived by I/O-bound applications. To mitigate such impact while retaining the benefit of CPU sharing, we introduce a new class of VMs called latency-sensitive VMs (LSVMs), which achieve better performance for I/O-bound applications while maintaining the same resource share (and thus cost) as other CPU-sharing VMs. LSVMs are enabled by the low latency VM scheduler called vSlicer, a hypervisor level technique that schedules each LSVM more frequently but with a smaller micro time slice. vSlicer enables more timely processing of I/O events by LSVMs, without violating the CPU share fairness among all sharing VMs. Our evaluation of a vSlicer prototype in Xen shows that vSlicer substantially reduces net-

work packet round-trip times and jitter and improves application-level performance. For example, vSlicer doubles both the connection rate and request processing throughput of an Apache web server; reduces a VoIP servers upstream jitter by 62%; and shortens the execution times of Intel MPI benchmark programs by half or more.

1.2.3 Offloading IRQ Processing to a Turbo-sliced Dedicated Core

In a virtual machine (VM) consolidation environment, it has been observed that CPU sharing among multiple VMs will lead to I/O processing latency because of the CPU access latency experienced by each VM. In our I/O optimization framework, vTurbo aims to accelerate I/O processing for VMs by offloading I/O processing to a designated core. More specifically, the designated core – called turbo core – runs with a much smaller time slice (e.g., 0.1ms) than the cores shared by production VMs. Most of the I/O IRQs for the production VMs will be delegated to the turbo core for more timely processing, hence accelerating the I/O processing for the production VMs. Our experiments show that vTurbo significantly improves the VMs network and disk I/O throughput, which consequently translates into application-level performance improvement.

1.2.4 Enabling Direct Read to HDFS Datanode VM Image

With its unlimited scalability and on-demand access to computation and storage, a virtualized cloud platform is attracting the big data systems such as Hadoop. However, virtualization introduces a significant amount of overhead to I/O intensive applications due to device virtualization and VMs or I/O threads scheduling delay. In particular, device virtualization causes significant CPU overhead as I/O data needs to be moved across several protection boundaries. We observe that such overhead especially affects the I/O performance of the Hadoop distributed file system (HDFS). In fact, data read from an HDFS datanode VM must go through virtual devices multiple times – incurring non-negligible virtualization overhead – even though both client VM and datanode VM may be running on the same machine. Our proposed vRead, a programmable framework which connects

I/O flows from HDFS applications directly to their data. vRead enables direct reads to the disk images of datanode VMs from the hypervisor. By doing so, vRead can significantly avoid device virtualization overhead, resulting in improved I/O throughput as well as CPU savings for Hadoop workloads and other applications relying on HDFS.

1.3 Dissertation Organization

This dissertation is organized as follows: chapter 2 discusses the design and implementation of vSlicer which reduces VM scheduling delay hence the I/O processing delay achieved by applications. Chapter 3 discusses vTurbo which offloads VM IRQ processing to a turbo core so that improves network and disk I/O throughput. Chapter 4 discusses the vRead enabling a ”direct” read to HDFS datanode VM’s disk image to reduce the data movement overheads in virtualized Hadoop clusters. Chapter 5 presents the related work in my area. We conclude the dissertation in Chapter 6 and discuss the future work in Chapter 7.

2 LATENCY-AWARE VIRTUAL MACHINE SCHEDULING VIA DIFFERENTIATED-FREQUENCY CPU SLICING

Recent advances in virtualization technologies have made it feasible to host multiple virtual machines (VMs) in the same physical host and even the same CPU core, with fair share of the physical resources among the VMs. However, as more VMs share the same core/CPU, the CPU access latency experienced by each VM increases substantially, which translates into longer I/O processing latency perceived by I/O-bound applications. To mitigate such impact while retaining the benefit of CPU sharing, we introduce a new class of VMs called *latency-sensitive VMs (LSVMs)*, which achieve better performance for I/O-bound applications while maintaining the same resource share (and thus cost) as other CPU-sharing VMs. LSVMs are enabled by *vSlicer*, a hypervisor-level technique that schedules each LSVM more frequently but with a smaller micro time slice. vSlicer enables more timely processing of I/O events by LSVMs, without violating the CPU share fairness among all sharing VMs. Our evaluation of a vSlicer prototype in Xen shows that vSlicer substantially reduces network packet round-trip times and jitter and improves application-level performance.

2.1 Introduction

The advent of the cloud computing paradigm has allowed enterprises and users to reduce their capital and operational expenditures significantly, because they can simply lease cloud resources to host their applications with a simple pay-as-you-go charging model. A key approach that powers cloud-based hosting is virtual machine (VM) consolidation, where a single physical machine is "sliced" into multiple VMs each assigned virtual core(s) for their execution.

While each VM is typically assigned at least one virtual core (*e.g.*, vCPU in Xen [13] parlance), the mapping between virtual and physical cores is not always one-to-one. For example, in commercial cloud offerings such as Amazon EC2 [1], the compute instances (VMs) are allocated in the units of EC2 compute units (ECU), each of which is roughly equivalent of a 1GHz machine, with the smallest EC2 instance allocated 1 ECU. In a 3 GHz physical machine, there may be three VMs sharing a physical CPU (pCPU). In such cases, the CPU scheduler in the underlying hypervisor (*e.g.*, Xen's default credit scheduler) schedules the runnable VMs in a round-robin fashion, with each VM given access to the physical CPU for the same amount of time, ensuring fairness among the CPU-sharing VMs.

Unfortunately, recent research [4–6, 8, 14] has discovered a serious downside of CPU sharing among multiple VMs: It leads to significant negative impact on I/O-bound applications running in those VMs. In this paper, we especially address a key aspect of the impact: *I/O processing latency perceived by applications.* More specifically, a VM with a pending I/O event will have to wait for its turn to access the CPU before processing the I/O event. Because of the multiple sharing VMs, the CPU access latency tends to be a *multiple* of the default CPU time slice for each VM (*e.g.*, 30ms in Xen); and such latency cannot be hidden from the corresponding application. This impact is particularly harmful to I/O-bound applications, which in this paper refer to applications *involving both I/O and computation, with I/O dominating computation*. For example, consider a simple VoIP gateway server which basically establishes and maintains connections between clients. For fast call setup and traffic relay, the gateway's network I/O dominates its computation (e.g., audio transcoding). With default CPU slices for the sharing VMs, the VM that hosts the gateway may not be able to access the CPU in time to process requests for new calls or traffic from ongoing calls. Another example is a low-volume web server that needs to quickly respond to client requests, yet its overall CPU usage is relatively lower.

To avoid the impact on I/O processing latency, one could choose to request a non-sharing VM that exclusively occupies a physical CPU. However, that would incur higher cost which may not be desirable for cost-sensitive customers. In this paper, we propose to mitigate such impact *with the presence of CPU-sharing VMs* (*e.g.*, small- or micro-

instances of EC2). More specifically, we introduce a new (sub)class of VM instances called *latency-sensitive VMs* (LSVMs), which will achieve better performance for I/O-bound applications. Contrary to LSVMs, we also define non-latency-sensitive VMs (NLSVMs) for the execution of *CPU-bound applications that do not have stringent timing/latency requirement*. LSVMs and NLSVMs will share the same CPU with fair share and similar cost; whereas the LSVMs will achieve lower I/O processing latency.

One way to enable LSVMs, as advocated by existing work [15–18], is to modify the hypervisor's CPU scheduler to prioritize certain I/O-bound VMs over the CPU-bound ones. For example, [15] preferentially schedules communication oriented applications over their CPU-intensive counterparts. Unfortunately, it introduces short-term unfairness in CPU shares. Similarly, partial boost is used in [16] to help I/O-bound tasks to preempt a running vCPU in response to an incoming event. However, such a system is hard to configure for preserving fairness among the sharing VMs, which is undesirable for a VM-hosting cloud. The credit scheduler is extended in [17] to support soft real-time applications. But it may give more CPU time to latency-sensitive VMs thus breaking the fairness among VMs.

In this paper, we propose our solution named *vSlicer* to realize LSVMs. vSlicer is based on a simple idea which we call *differentiated-frequency microslicing*. Traditional VM schedulers such as Xen's credit scheduler "slice up" a CPU in relatively large time slices. Under vSlicer, we further divide a CPU slice (*e.g.*, 30ms) of a given LSVM into several *microslices* (*e.g.*, 5ms) and schedule the LSVM at a higher frequency (*e.g.*, 6 times) compared to an NLSVM (one time) in each scheduling round. Therefore, both the LSVMs and NLSVMs sharing a physical core will still obtain the same amount of CPU time thus ensuring fairness; but an LSVM will be scheduled more frequently albeit with a smaller time slice, resulting in shorter CPU access latency for the LSVM. Consequently, for an I/O-bound application, vSlicer gives the corresponding LSVM more frequent CPU accesses – each for a shorter duration – to process its pending I/O activities, resulting in better application-level performance.

Since the overall CPU share is the same for both LSVMs and NLSVMs, their charging model does not need any change and can be priced the same. At first glance, it may appear

that every cost-sensitive customer (namely, one who is unwilling to upgrade to VMs with exclusive CPUs) would request only LSVMs. This is not true for the simple reason that LSVMs may not help all applications across the board. In particular, running a CPU-bound application in an LSVM may actually be worse than running it in an NLSVM, because of the more frequent context switches and subsequently more frequent cache flushes. Therefore, customers running CPU-bound applications will be motivated to choose NLSVMs over LSVMs. Consequently, we are likely to see a mix of LSVMs and NLSVMs sharing the physical machines.

2.2 Problem Demonstration and Motivation

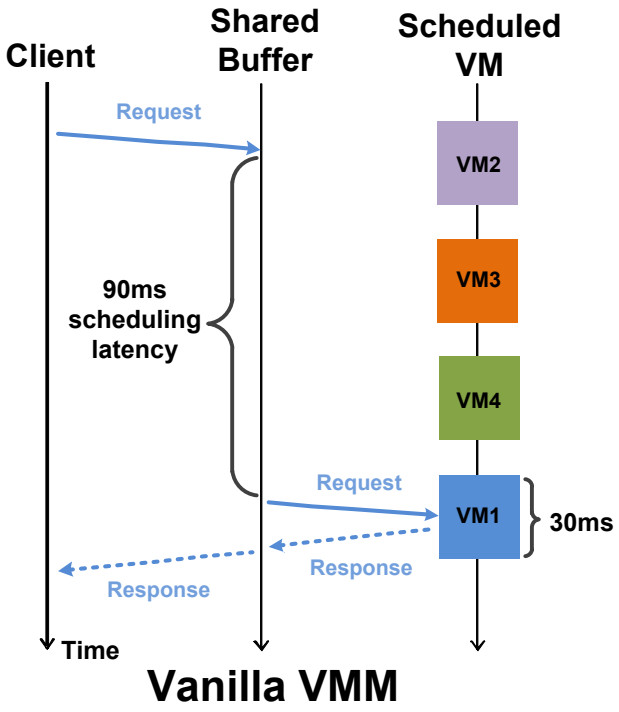


Figure 2.1.: Application responsiveness with credit scheduler

In this section, we motivate the problem by demonstrating the impact of VMs' CPU sharing on I/O processing latency. We then discuss the inadequacy of existing solutions.

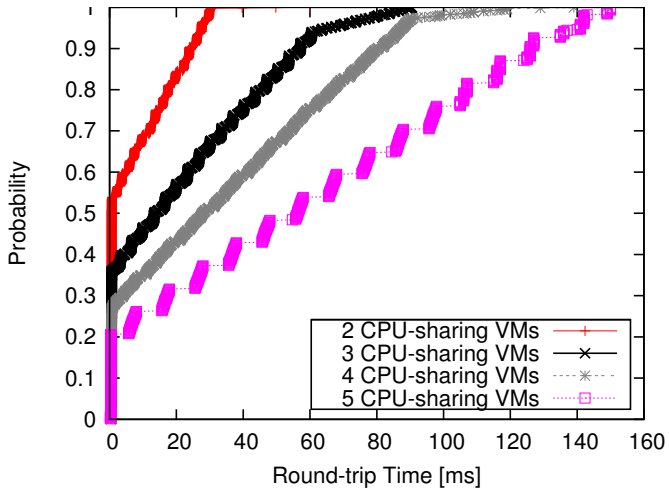


Figure 2.2.: CDF of *ping* round-trip time

2.2.1 Impact of CPU Sharing

To understand the negative impact of VM CPU sharing on the latency of I/O processing, consider the example shown in Figure 2.1. In this example, 4 VMs are sharing a physical CPU. VM1 is hosting an I/O-bound application while VM2-VM4 are hosting CPU-bound applications. The application in VM1 waits for client requests and then responds to the requests with data or control messages. This simple communication pattern can be found in many applications such as web servers, VoIP proxies, and MPI jobs. We assume that the VM scheduler in the hypervisor uses a proportional-share scheduling policy adopted by many commercial VM platforms (e.g., Xen, that is used in Amazon EC2 [1], RackSpace [19] and GoGrid [20] commercial clouds). Since each VM has a runnable task in it, it occupies the entire CPU slice allotted to it. As shown in the figure, when a request for VM1 arrives at the physical host, it needs to be buffered outside VM1 (e.g., in the VMM or in the privileged driver domain not shown in the figure), until VM1 is scheduled to run. When VM1 gets scheduled, it will process the request and generate a response. Assuming a CPU slice of 30ms, the request response latency can be as high as 90ms (*i.e. (Number of sharing VMs -1)* $\times$ *Time Slice*). Such a high latency hampers the responsiveness (and consequently, request processing rate) of the application in VM1.

We perform a simple experiment to demonstrate this increase in latency empirically. Figure 2.2 shows the CDF of the round-trip time (RTT) by "pinging" VM1. In our measurement experiments, we vary the number of non-idle, CPU-sharing VMs from 2 to 5 (including VM1). Our results clearly show that the ping RTT increases with the number of CPU-sharing VMs; and the worst-case RTT is proportional to *(Number of sharing VMs -1)* $\times$ *Time Slice*.

2.2.2 Problems with Alternative Solutions

We now examine several alternative solutions and argue why they do not work well in our setting.

Prioritize I/O-Bound VMs The first option to reduce the above I/O processing latency is to prioritize the VMs running the I/O-intensive applications. In fact, Xen's credit scheduler uses *BOOST* mechanism to shorten the I/O response time by temporarily boosting (*i.e.* assigning a higher priority to) the I/O bound VMs. This mechanism works quite well for pure I/O bound VMs. However, in the presence of heterogeneous workloads, once the VM gets scheduled to process the I/O request by the BOOST mechanism, it will consume its CPU share (*i.e.* credits in Xen terms) due to the CPU bound segment of the workload. This will effectively disable the BOOST mechanism for the rest of this scheduling cycle resulting in higher I/O latency. In other words, while BOOST can temporarily cede the CPU to I/O-bound VMs, it can often lead to exhausting the VM's credits early and, as a result, the VM may starve for the rest of the scheduling round (since the credit scheduler is CPU-fair across VMs).

A naive workaround to this would be to aggressively boost the I/O bound VMs without considering its CPU share. Unfortunately, this prioritization will break the overall CPU fairness in the system. We demonstrate such an unfairness in Figure 2.3, where VM1 is hosting an I/O-bound application with a network-intensive task and a computation task whereas other VMs are hosting computation-intensive applications. The incoming packets to VM1 trigger the boosting of VM1 so that it can process the packets. However, since the

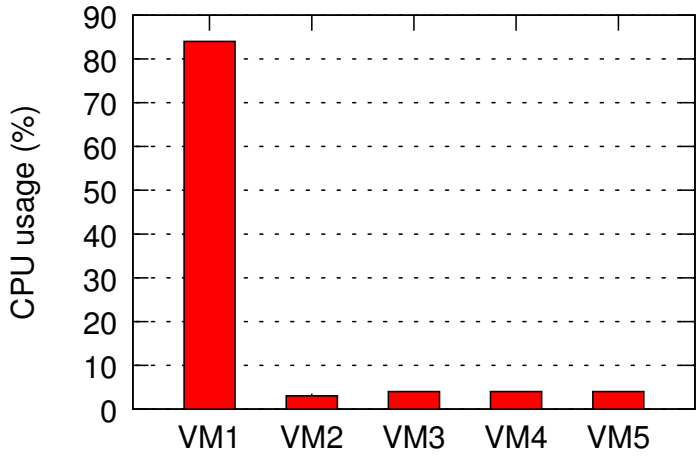


Figure 2.3.: Unfair CPU allocation under aggressive boost

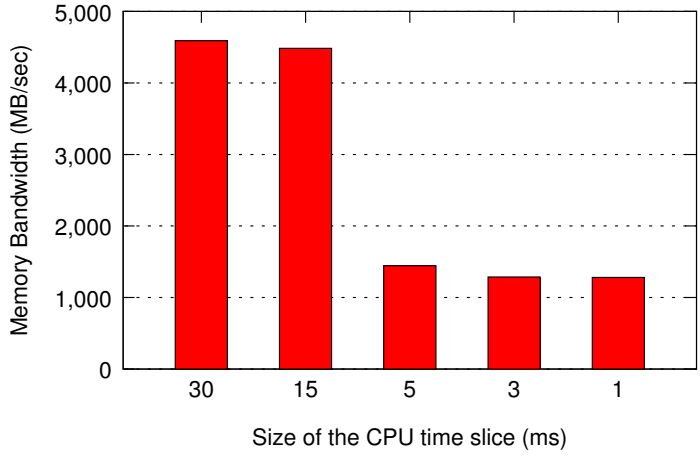


Figure 2.4.: STREAM benchmark performance under credit scheduler

hypervisor does not preempt a scheduled VM as long as the VM has runnable tasks, the computation portion of the application in VM1 will consume the rest of the time slice after the packet processing is done. This causes CPU-time deprivation of other VMs, as long as packets destined to VM1 keep arriving, compromising the CPU fairness of the overall system.

Soft Real-Time Scheduler The second option is to adopt a soft real-time scheduler such as Xen's former scheduler – Simple Earliest-Deadline First (SEDF) scheduler [21]. SEDF is based on a preemptive, deadline-driven real-time scheduling algorithm to achieve latency

guarantees. However, such a scheduler requires complex configuration and careful parameter tuning and selection – per-VM – to achieve the latency guarantees desired, which may not be possible in a cloud environment with dynamic placement and migration of VMs. In addition, and perhaps more importantly, extending SEDF to perform global load-balancing on multicore systems is non-trivial, making it not attractive on multicore platforms. Because of these reasons, SEDF has been replaced by the credit scheduler as Xen's default scheduler. Another low-latency scheduler available in Xen is based on Borrowed Virtual Time (BVT) scheduling scheme [22]. BVT achieves low latency by making use of virtual-time warping. However, lack of a non-work-conserving mode in BVT severely limits its usability in a number of environments, leading to its retirement from Xen's latest version.

Reducing Slice Size for all VMs The third option is to uniformly reduce the time slice size of the credit scheduler so that all the sharing VMs will get scheduled in and out more frequently, resulting in shorter CPU access latency. However, such an option would increase the number of context switches (and cache flushes) in the system, degrading the performance of CPU-bound applications running in the NLSVMs. To demonstrate the problem with this option, we measure the *memory bandwidth* of VMs running the STREAM benchmarks [23], scheduled by the credit scheduler under various time slice sizes (30ms to 1ms). The STREAM benchmarks measure memory bandwidth for large array operations such as copy, addition, scalar multiplication, and triad. Here we only present the "STREAM-copy" results in Figure 2.4. (We obtain similar results from the other 3 benchmarks.) The results indicate that reducing the time slice size uniformly is clearly not desirable as it degrades the memory access efficiency and consequently application performance of the VMs.

2.3 Design

The previous section suggests that, if the CPU-sharing VMs are scheduled in a strictly round-robin fashion, it will be difficult to reduce the I/O processing latency without hurting the performance of CPU-bound NLSVMs. On the other hand, prioritizing the LSVMs may violate the CPU share fairness among all VMs.

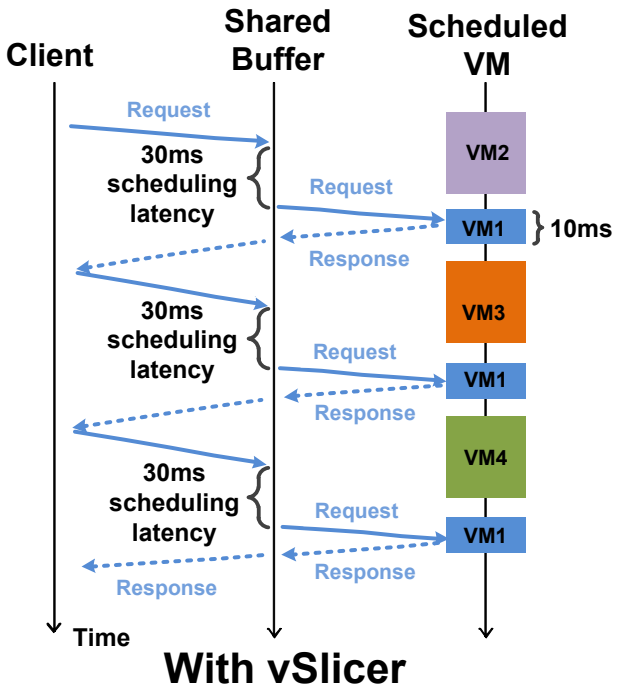


Figure 2.5.: Application responsiveness with vSlicer

To address this dilemma, we come up with the following key idea behind vSlicer: Within one scheduling round, the CPU time for an LSVM does not have to be allocated in one single time slice. Instead, it can be allocated "in installment" as long as the sum of the installments (i.e., microslices) is equal to a standard CPU time slice. Such a high-frequency microslicing will give more opportunities to the LSVM to process pending I/O events; yet it does not affect/preempt the regular time slices allocated to the NLSVMs. This ensures timely processing of I/O events while maintaining fair share of the CPU among all VMs. We illustrate this idea in Figure 2.5 for the same application scenario as in Figure 2.1. In one scheduling round, the LSVM (VM1) will be scheduled three times (instead of once), each for a microslice of 10ms (instead of 30ms). As a result, it can process three requests (instead of one) in the same time period, improving the application's responsiveness.

For the purely CPU-bound applications, as demonstrated in last Section, there is a strong incentive *not* to run them in LSVMs because the higher-frequency microslicing will cause more frequent cache flushes which will hurt application performance. Fortunately,

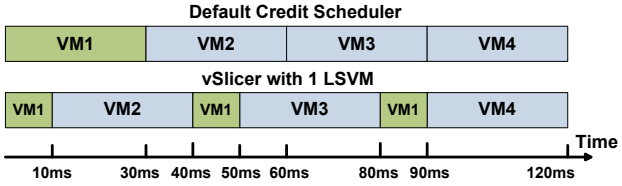


Figure 2.6.: vSlicer with 1 LSVM

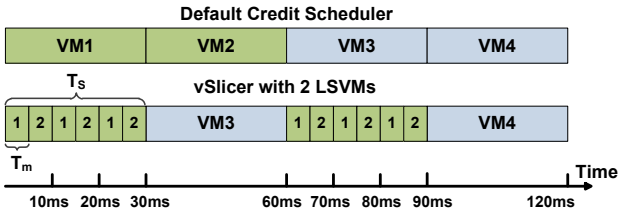


Figure 2.7.: vSlicer with 2 LSVM

Figure 2.8.: vSlicer scheduling sequence (The green block indicates LSVM)

the NLSVMs under vSlicer will give these applications the same performance as if running them in round-robin-scheduled VMs with the default time slice.

2.3.1 vSlicer Scheduling Model

The idea of CPU microslicing itself is quite general; one could pick any size for the microslice and simply derive the scheduling frequency. There are two main concerns one needs to keep in mind though. First, setting the microslice too small will excessively increase the context switch overhead; so it is important to keep it to a reasonable duration (*e.g.*, at least 5ms). Second, the best schedule one can come up with, in terms of latency for LSVMs, depends on the number of LSVMs and NLSVMs sharing a core. In practice, we expect only a small number (≤ 5) that share a core, and even among these, the number of LSVMs is going to be very small (≤ 3).

We use the following approach to determine the scheduling order in one scheduling round. Assume m LSVMs and n NLSVMs are sharing a single CPU core.

We denote the scheduling period (i.e., scheduling round) by T_P and the total time an LSVM executes during a scheduling period as T_{LSVM}. Similarly, the total time an NLSVM executes during a scheduling period is T_{NLSVM}. We want T_{NLSVM} to be a fairly large value to allow each CPU-bound VM to execute sufficiently long. (In our implementation we use Xen credit scheduler's default time slice 30ms as T_{NLSVM}.) Since we aim to fairly allocate the CPU among all the VMs (both LSVMs and NLSVMs), we want the following to hold:

$$T_{LSVM} = T_{NLSVM} \tag{2.1}$$

Let us denote the time period where one (micro-)round of LSVMs are scheduled after scheduling an NLSVM as T_S. vSlicer runs all the LSVMs during T_S in round robin fashion. We want to further divide T_S into micro time slices T_m (refer to Figure 2.7 for illustration of T_S and T_m; here subscript m indicates "micro" rather than the number of LSVMs). The selection of T_m depends on the scheduling latency we intend to achieve. We will further discuss the scheduling latency achieved by the vSlicer later in this section. Depending on the selection of T_m, an LSVM can run one or more times during a single time slice T_S. Let us denote the total time the i^{th} LSVM runs during T_S as T_{n_i}.

$$\sum_{i=1}^{m} T_{n_i} = T_S \tag{2.2}$$

Suppose the i^{th} LSVM can get scheduled r_i times during T_S. We have:

$$T_{n_i} = r_i \times T_m \text{ where } r_i \geq 1 \tag{2.3}$$

In this paper, we assume all the LSVMs have the same latency requirement and hence, for any $i, j \in \{1, m\}$ we have $T_{n_i} = T_{n_j} = T_n$ and $r_i = r_j = r$. Equation 2.3 becomes

$$T_n = r \times T_m \text{ where } r \geq 1 \tag{2.4}$$

and

$$T_S = m \times T_n \tag{2.5}$$

Given vSlicer's alternating scheduling of LSVMs and NLSVMs (i.e., it schedules a round of all LSVMs followed by one of the NLSVMs), the total time that an LSVM executes during a scheduling period T_P is equal to the number of NLSVMs multiplied by the time an LSVM executes during a time slice T_S (*i.e.* T_n). That is:

$$T_{LSVM} = n \times T_n \tag{2.6}$$

A scheduling period consists of running times of all LSVMs and NLSVMs and therefore we get:

$$\begin{aligned} T_P &= mT_{LSVM} + nT_{NLSVM} && (2.7)\\ &= m \times (n \times T_n) + nT_{NLSVM} && (2.8) \end{aligned}$$

Rearranging the first term of RHS of Equation (2.8) and substituting from Equation (2.5) gives us:

$$T_P = nT_S + nT_{NLSVM} \tag{2.9}$$

Also substituting for T_{LSVM} from (2.1) to (2.7) we get :

$$\begin{aligned} T_P &= mT_{NLSVM} + nT_{NLSVM} \\ &= (m+n)T_{NLSVM} && (2.10) \end{aligned}$$

Combining Equations (2.9) and (2.10) gives us an important invariant we maintain in the system:

$$nT_S + nT_{NLSVM} = (m+n)T_{NLSVM} \tag{2.11}$$

That is, maintaining this invariant ensures that we are not violating CPU share fairness while scheduling LSVMs more frequently. Moreover, Equation (2.11) allows us to define T_S, T_n in terms of T_{NLSVM}. That is :

$$\begin{aligned} T_S &= \frac{mT_{NLSVM}}{n} \\ T_n &= \frac{T_{NLSVM}}{n} \\ T_m r &= \frac{T_{NLSVM}}{n} && (2.12) \end{aligned}$$

As mentioned earlier, the selection of T_m depends on the desired scheduling latency of the LSVM. Equation (2.12) defines the product of T_m and r in terms of T_{NLSVM} and n. The only restriction for the selection of T_m is, it should be a whole divisor of $\frac{T_{NLSVM}}{n}$. However, selecting a too small value for T_m will increase the number of context switches during T_S, affecting the performance of the all LSVMs.

Let us denote the required latency for an LSVM during T_S as T_l. To achieve this scheduling latency we should schedule the i^{th} VM within T_l. Since we schedule all the LSVMs in a round-robin order, all the other $(m-1)$ LSVMs should be executed in less than T_l. That is:

$$(m-1)T_m \leq T_l$$

which gives us the upper bound for T_m:

$$T_m \leq \lfloor \frac{T_l}{(m-1)} \rfloor$$

If we consider the influence of NLSVMs, the scheduling latency curve for a specific LSVM looks like a *continuous wavy line*. The wave crest is $\mathrm{T}_{NLSVM} + (m-1)T_m$.

Examples We now show two examples of scheduling sequence under two different settings. Figure 2.8 illustrates two scheduling sequences for a system running four VMs. In Figure 2.6, we have one LSVM and three NLSVMs. If all these VMs were scheduled by the default credit scheduler, any of them would experience a 90ms scheduling latency. Under vSlicer, by dividing the time slice of the LSVM (*i.e.* VM1) to multiple microslices and scheduling it three times during the scheduling round, the latency drops to 30ms. In Figure 2.7, there are two LSVMs and two NLSVMs in the system. By dividing the time slice into 5ms microslices, vSlicer can achieve a best-case latency of 5ms and a worst-case latency of 35ms.

In our discussion towards the end of motivation section, we emphasized that reducing the time slice uniformly for all sharing VMs is not a desirable option, primarily due to the increased context switches between the VMs. Now that we have discussed the details of vSlicer, let us quantitatively compare the credit scheduler – with uniformly reduced time

slice – with vSlicer using a system with two LSVMs and two NLSVMs. With the credit scheduler having the default time slice, the CPU access latency of each LSVM is (m+n-1)T_{NLSVM}. Here it is (4-1)$T_{NLSVM} = 3 \times 30 = 90ms$. In order to reduce the latency to 15ms, we need to reduce the time slice from 30ms to 5ms, which will make the context switch rate increase by 6×. With vSlicer, however, setting $T_m = 5ms$ – to achieve 15ms average latency – would increase the number of context switches only by 3×.

2.4 Implementation

vSlicer only requires a simple modification to the VM scheduler in the hypervisor. The VMs in the physical host are grouped at two levels. First, vSlicer maintains a list of VMs that are executing in a physical CPU. Second, within this group vSlicer divides these VMs into LSVMs and NLSVMs. Decision on whether a particular VM is LSVM or NLSVM is left to the user (or the cloud administrator) and vSlicer provides an interface to the administrative tools (such as *xm tools* in Xen) to configure that. If dynamic VM characterization is preferred, existing methods using virtual interrupt counters or pending packet counters can be applied to infer VM's type dynamically. However, the grouping of VMs per physical CPU is done by the global load balancing algorithm of the VM scheduler.

While the design of vSlicer is generic and hence applicable to many VMMs (*e.g.*, Xen, VMware [24]), we implement a prototype of vSlicer in Xen 3.4.2. In our implementation, we add a new scheduler type in Xen, called sched_vSlicer by extending the credit scheduler. The vSlicer code is in the critical path of the scheduler code which is frequently executed. Therefore we keep the modifications to the critical path of the credit scheduler to a minimum, with only 250 lines of additional code. The user-level utilities add another 400 lines of code which is executed only when the user configures the system using the Xen management tools. vSlicer does not depend on para-virtualization for its scheduling function. So our prototype can support Xen HVM guests without modifications or performance degradation.

Since vSlicer is based on the credit scheduler, vSlicer inherits its proportional fairness policy and multi-core support. We maintain the credit scheduler's existing set of controls, *weight* and *cap*, that decide the proportional share of the VM, and the maximum amount of CPU a domain will be able to consume even if the host system has idle CPU cycles respectively. We add a new control in addition to these two to specify the micro time slice. Initially vSlicer treats all the VMs as NLSVMs, which have their micoslices set to zero. When a user configures a particular VM to be LSVM, the microslice of that VM will be set to the specified value. This action will trigger vSlicer configuration functions, which will in turn recalculate the global parameters such as T_S. Starting from the next scheduling interrupt, vSlicer will schedule that VM as an LSVM.

Scheduling Algorithm The most important function that we modify is *do_schedule*, which is executed in the critical path and responsible for selecting the next vCPU for pCPU from the run queue. We show the pseudo-code of the algorithm in Algorithm 1.

We assign *micro credits* to each LSVM in addition to the credits assigned by the original algorithm of Xen credit scheduler. vSlicer algorithm uses the micro credits to schedule LSVMs during T_S in a round-robin order. We initialize the algorithm by initializing T_{NLSVM}, T_S, and T_m. T_{NLSVM} is defined by the implementation (in our implementation we used Xen's default 30ms). T_S and T_m can be calculated using T_{NLSVM}, m, n, and equations in Section 2.3.1. This initialization has to be done in the event of: a vCPU migration (for load balance on multi-core), a VM initialization, a VM shutting down or any other event that changes the number of VMs running on the particular CPU core.

vSlicer algorithm is executed whenever the time slice of the currently running VM expires. First the algorithm checks the VM type. If it is an NLSVM, the time slice of it has expired and hence the VM is inserted to the back of the *run queue*. In vSlicer both NLSVMs and LSVMs share a single run queue. If the the current VM is an LSVM, depending on how much credits and *micro credits* the VM has, it will be scheduled to run in the same T_S, in the same T_P, or in the next scheduling period. Then the algorithm picks the next VM to run from the head of the run queue. If it is an NLSVM, it will be assigned a regular time slice (T_{NLSVM}). If it is an LSVM, it will be assigned a microslice.

Algorithm 1 Scheduling Algorithm for vSlicer

Require: $num\_nlsvm \geq 1$

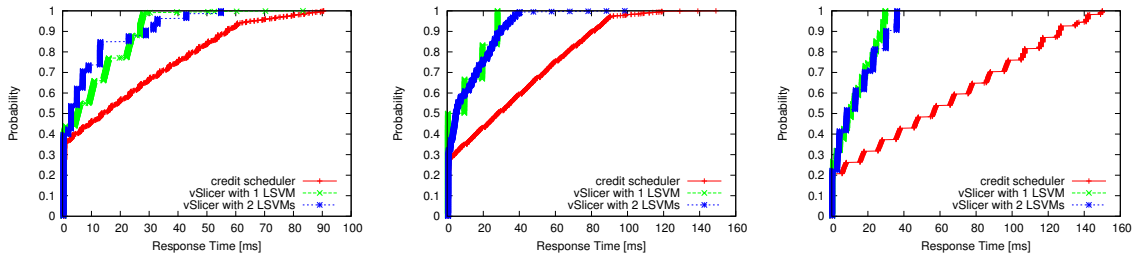
Require: $num\_lsvm + num\_nlsvm \geq 3$

Ensure: $schedule\_time = now$

Ensure: $time\_slice = T_{NLSVM}$

Ensure: $micro\_slice = T_m$

```
1:  burn_credit(curr_vm.schedule_time, now)
2:  if curr_vm is nlsvm then
3:      insert_tail(curr_vm, runq);
4:  else[curr_vm is lsvm]
5:      burn_micro(curr_vm.micro_credits, micro_slice)
6:      if curr_vm.credits > 0 then
7:          if curr_vm.micro_credits > 0 then
8:              insert_before_nlsvm(curr_vm, runq);
9:          else[curr_vm.micro_credits ≤ 0]
10:             insert_after_nlsvm(curr_vm, runq);
11:         end if
12:     else[curr_vm.credits ≤ 0]
13:         insert_tail(curr_vm, runq);
14:     end if
15: end if
16: next_vm ⇐ get_first_elem(runq);
17: if next_vm is nlsvm then
18:     next_vm.runtime ⇐ time_slice;
19: else[next_vm is lsvm]
20:     next_vm.runtime ⇐ micro_slice;
21: end if
22: run(next_vm);
```



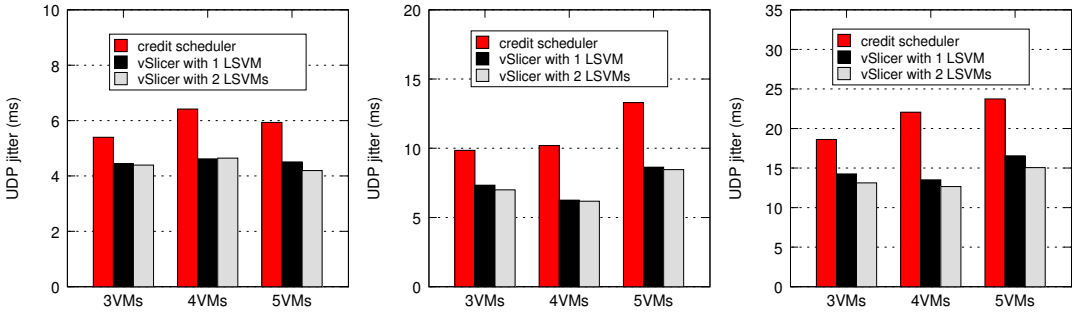
(a) 3 non-idle VMs sharing a core (b) 4 non-idle VMs sharing a core (c) 5 non-idle VMs sharing a core

Figure 2.9.: CDFs for RTTs of 100 *ping* packets under default credit scheduler and vSlicer

2.5 Evaluation

In this section, we present our detailed evaluation of vSlicer using the Xen-based prototype. We use both micro-benchmarks and application-level benchmarks to evaluate the effectiveness of vSlicer. Our experiments evaluate three key aspects: (a) transport-level latency reduction achieved by vSlicer; (b) overall CPU-sharing fairness with vSlicer; and (c) application-level performance improvement by vSlicer.

Experimental Setup Our experiments involve physical machines (desktops as clients and servers as VM hosts) connected by a Gigabit Ethernet network. Each physical server hosts multiple VMs and has a dual-core 3GHz Intel Xeon CPU with 4GB of RAM and a Broadcom NetXtreme 5752 Gigabit Ethernet card. These hosts run Xen 3.4.2 with Linux 2.6.18 running in the driver domain (dom0). The VMs share one core of the host, whereas the driver domain is pinned to the other core. Each VM in this host is allocated 512MB of RAM and a single vCPU, except the VM that hosts the MyConnection media server (Section 2.5.2) which is allocated 1GB RAM following the requirement of the MyConnection benchmarks. The physical client machine has a 2.4GHz Intel Core 2 Duo CPU with 4GB of RAM and an Intel Pro Gigabit network card and runs Linux 2.6.35.



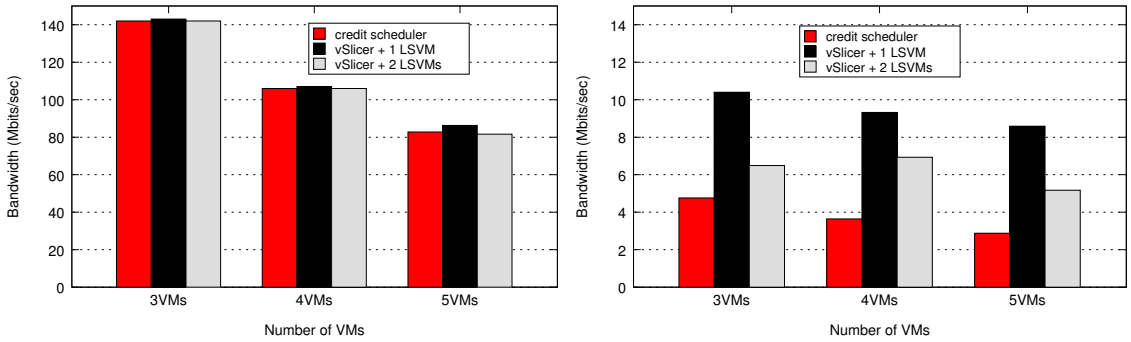
(a) Jitter for 256B datagrams (b) Jitter for 512B datagrams (c) Jitter for 1024B datagrams

Figure 2.10.: Effect of vSlicer on UDP jitter

2.5.1 Evaluation with Micro-benchmarks

This section presents improvement of network I/O performance achieved by vSlicer using micro-benchmarks. In each experiment we vary the number of VMs sharing the same core from 3 to 5 and measure the same transport-level metrics under vSlicer and Xen's default credit scheduler, respectively. We keep the CPU utilization of each VM to 40% using the *lookbusy* tool [25].

Ping RTT Recall the experiment presented in motivation section that measures the RTTs of ping packets to a non-idle VM from another physical machine in the same LAN. We repeat the same experiment, but use vSlicer as the VM scheduler and compare the results with those achieved by the default scheduler. Figure 2.9(a) and Figure 2.9(b) shows the CDFs of RTTs of 100 *ping* packets, with 3, 4, and 5 CPU-sharing VMs, respectively. For each setup, we show the CDFs under the credit scheduler, vSlicer with 1 LSVM (the ping receiver), and vSlicer with 2 LSVMs (one being the ping receiver), respectively. These results show that vSlicer consistently reduces the ping RTTs in all setups. For example, in the 4-VMs scenario (Figure 2.9(b)), vSlicer reduces the average RTT from 35ms to 10ms with 1 LSVM (the other three are NLSVMs), a 71% reduction. With 5 CPU-sharing VMs (Figure 2.9(b)), the average ping RTT is shortened by about 80% under vSlicer. More importantly, we find that, under vSlicer, the RTT towards an LSVM *does not increase linearly with the number of sharing VMs*. With vSlicer, the average RTT we observe across



(a) In LAN (b) For high RTT setting

Figure 2.11.: Effect of vSlicer on TCP throughput

our experiments remains about 12ms with 1 LSVM; and 14ms with 2 LSVMs; whereas a near-linear increase in average RTT is observed under the default scheduler.

UDP Jitter UDP is a simpler transport protocol with no reliable, in-order packet deliver guarantee. Yet UDP is popular in audio/video streaming, online gaming and other latency-sensitive applications. We measure the jitter of UDP datagrams, which will translate into user-level QoS of the aforementioned applications. We use Iperf [26] to generate a stream of UDP datagrams and vary the datagram size in each setup. The UDP receiver runs in a non-idle VM (an LSVM when running on vSlicer) and the UDP sender is a different physical machine in the same LAN. We also vary the number of CPU-sharing VMs from 3 to 5. The average UDP jitter observed on the receiver side is shown in Figure 2.10. The results under different datagram sizes all show UDP jitter reduction. The reason for the jitter reduction is that an LSVM has multiple opportunities to run during one scheduling round under vSlicer (vs. only one under the default scheduler), leading to more timely and more evenly timed processing of UDP datagrams.

TCP Throughput Our measurement of TCP throughput generates some interesting (and somewhat surprising) results. Since vSlicer reduces a VM's CPU access latency and benefits latency-sensitive applications, we first thought that vSlicer would also improve TCP throughput to/from a VM. We use Iperf to measure the TCP bandwidth between a physical machine and a VM in the same LAN. The Iperf server runs in a non-idle (40%

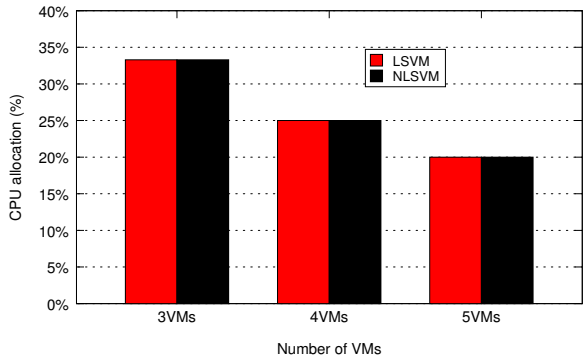


Figure 2.12.: Average CPU utilization for the two types of VMs under vSlicer

CPU load) VM sharing the CPU core with 2-4 other non-idle VMs. Interestingly, as shown in Figure 2.11(a), vSlicer does not improve TCP throughput within a LAN. The reason, after a closer examination, is the following: First, even with vSlicer, LSVMs experience longer latencies periodically when the NLSVMs are getting scheduled. This delay would be less compared to the delay with the default credit scheduler (30ms compared to the 60ms in 3 VM scenario). However, this is still high compared to the sub-millisecond latencies in the LAN environment. Second, when we microslice the time slice of the LSVM (in this case from 30ms to 15ms) , we also reduce the amount of packets that can be processed during a single micro time slice by some fraction (by 50% in this case), which means that the rest of the packets have to wait one full NLSVM execution time slice until they get processed, which makes throughput of the connection similar to that achieved by the credit scheduler.

However, the results are different when we look at a WAN environment. We simulate higher RTTs in a WAN by adding 30ms of network delay between the TCP sender and receiver using Linux *netem* module. The 30ms additional delay is based on average RTTs between our lab and well-known services (e.g. Google, AmazonEC2 and Microsoft Azure). This time we observe that vSlicer improves TCP throughput by up to 3×, as shown in Figure 2.11(b). When we add 30ms network delay, this delay will effectively mask the execution period of the NLSVM. Recall that our VM scheduling pattern from Section 4.3 – an execution of an NLSVM is always followed by an execution period of all the LSVMs.

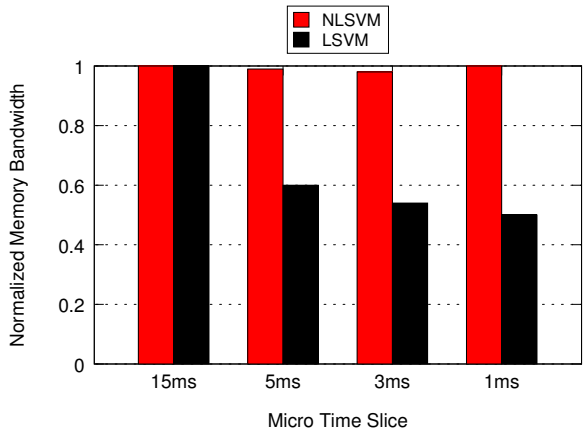
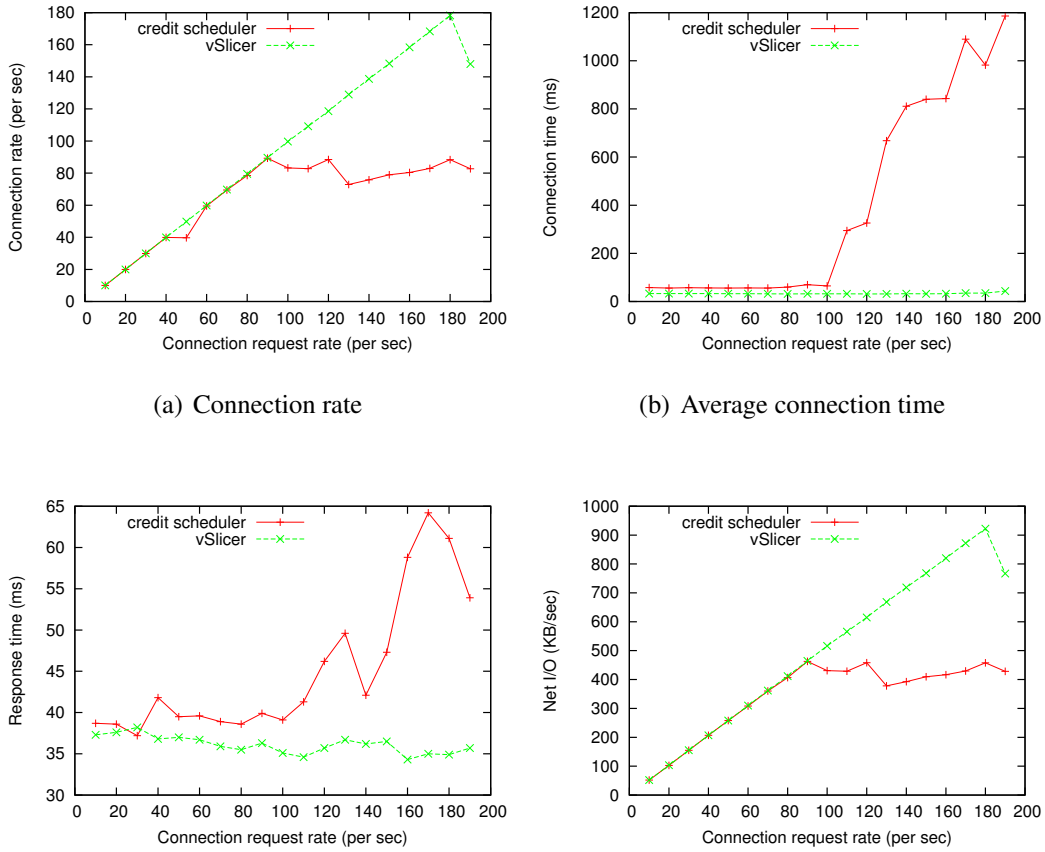


Figure 2.13.: STREAM benchmark performance under different configurations

So if we consider 3 VM case with one LSVM, once LSVM acknowledges a set of TCP packets and schedule out, it will take another 30ms time for the arrival of another batch of TCP data segments due to the added network delay. Now, during this 30ms in the receiving host, one NLSVM will be executed and the LSVM will be scheduled by the time of the arrival of TCP data packets, which can be immediately processed. On the other hand, if we consider the default credit scheduler, adding 30ms network delay will mask the execution time of just one VM. Since the credit scheduler schedules VMs in a round-robin fashion, in the 3 VM scenario, packets still have to wait one more time slice until the receiving VM gets scheduled. With the same experiment setting, we confirm that vSlicer can improve wide-area TCP throughput under varying additional delays (20ms-100ms) for the same reason.

Fairness of CPU Sharing After evaluating vSlicer's improvement of network I/O, we now evaluate the fairness of CPU sharing among all sharing VMs (LSVMs and NLSVMs). We use *xentop* to monitor the CPU utilization of each VM while running lookbusy and sysbench benchmark in each VM. We observe that, regardless of its type (LSVM or NLSVM), each VM has an equal share of the CPU as the other VMs. Figure 2.12 shows the average CPU utilization (reported by *xentop* over a period of 30 seconds) of one LSVM and one



(a) Connection rate

(b) Average connection time

(c) Average response time

(d) Net I/O

Figure 2.14.: Apache web server experiment results

NLSVM (out of a total of 3, 4, or 5 VMs) under vSlicer. The results show that vSlicer maintains CPU sharing fairness between the two types of VMs.

We then measure the performance of a CPU/memory-bound application running in an NLSVM under vSlicer. We use the STREAM benchmarks as in motivation section and run 4 VMs – two LSVMs and two NLSVMs in a physical host. We run the STREAM benchmark in one of the NLSVMs, each getting one regular 30ms time slice in a scheduling round, while we vary the microslice size (from 15ms to 1ms) of the sharing LSVMs. Figure 2.13 shows the results in terms of memory bandwidth achieved by the benchmark. For comparison, we normalize the memory bandwidth relative to the one achieved by the

default credit scheduler with the same 4 VMs and same workloads. The results show that the performance of STREAM running in the NLSVM (the red bars) is not affected by the more frequent scheduling of the LSVMs under vSlicer, maintaining (almost) the same performance as under the credit scheduler. To demonstrate the unsuitability of LSVMs for CPU-bound applications, we also run the STREAM benchmark in an LSVM and the results are shown by the black bars in Figure 2.13. This time the STREAM performance degrades with the decrease of microslice size (i.e., with the increase of LSVM scheduling frequency).

2.5.2 Evaluation of Application Performance

Experiment with Apache Web Server We first use the Apache web server along with httperf [27] to evaluate the effectiveness of vSlicer for I/O-bound applications. While not a soft-real-time application, the Apache web server is sensitive to (network and disk) I/O processing latency, which will cause delay in both connection establishment and data transmission stages and thus affect the web server's response time and request handling throughput.

In this experiment the physical server hosts four core-sharing VMs. Two of the VMs are LSVMs, with one of them running the Apache web server. A physical client machine generates requests for a 5KB web page with httperf to measure the web server's performance. To simulate the WAN environment, a random delay between 20ms to 40ms using the Linux *netem* is added. For comparison, we perform the experiment under the default credit scheduler and under vSlicer. We measure the following metrics: (a) connection rate, (b) connection time, (c) response time, and (d) net I/O (average network throughput), with the corresponding results shown in the four sub-figures of Figure 2.14. Under the credit scheduler, the connection rate saturates at 90 connections/sec and the net I/O throughput saturates at 450 KB/s. Under vSlicer, Apache can sustain up to a 180 connections/sec connection rate and achieve up to 900 KB/s throughput. Moreover, the connection time and response time are much shorter and more stable under vSlicer; whereas under the credit

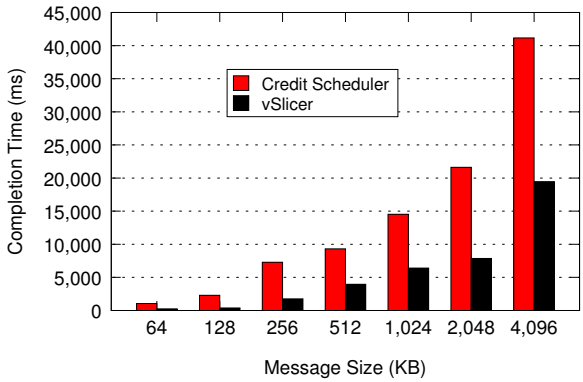


Figure 2.15.: Performance of Intel MPI benchmark: *Alltoall*

scheduler, these two metrics increase rapidly once the request rate goes beyond 100 requests/sec.

To understand the root cause for the saturated connection rate of 90 connections/sec under the credit scheduler, we first traced packets using *tcpdump* at multiple points: (1) in the client host, (2) in the driver domain of the physical server, and (3) in the LSVM where the Apache server runs. We make two interesting observations: First, when the connection rate goes beyond 90 connections/sec, packet retransmissions start to appear in the trace. Second, our further analysis of flows with packet retransmissions shows that almost all of the retransmissions happen due to the packets dropped at the driver domain (by comparing the traces from the driver domain and from the VM).

To identify the main culprit of the dropped packets inside the driver domain, we inserted tracing points along the path taken by the packets inside the driver domain from physical NIC (*peth*) to the VMs virtual interface (*vif*). We found out that the I/O ring buffer, which connects the driver domain and the VM, gets full when the request rate exceeds 90 connections/sec while the VM is waiting in the run queue. This in turn back-pressures the packet processing *tasklets* in the driver domain causing packet drops. On the other hand, with vSlicer, the LSVM running the Apache server gets scheduled more frequently and hence, it empties the ring buffer more often hence eliminating the back-pressure. Compared with the maximum CPU access latency (90ms) under the credit scheduler, the maximum latency for

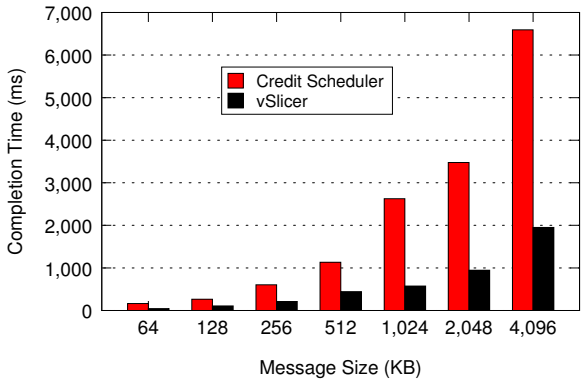


Figure 2.16.: Performance of Intel MPI benchmark: *Sendrecv*

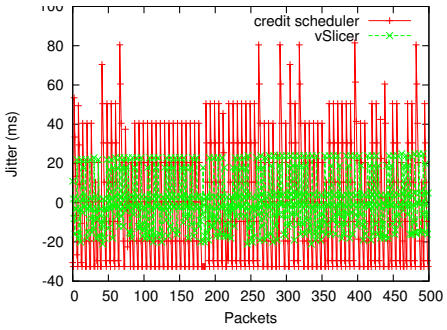


Figure 2.17.: Single line VoIP upstream jitter

the LSVM is 40ms under vSlicer. This translates into a higher connection rate (up to 180 connections/sec) of the web server without packet drops and retransmissions in the driver domain.

Experiments with MPI Benchmarks We next evaluate the effectiveness of vSlicer for reducing the execution time of MPI communication primitives using the Intel MPI Benchmark (IMB) [28]. Our setup consists of 4 VMs each with MPICH2 [29] libraries installed. We host these 4 VMs in two physical hosts with 2 VMs sharing a single CPU core. We also run 2 other VMs per core with CPU-bound tasks. When experimenting with vSlicer, we mark the VMs running the IMB as LSVMs and the VMs running CPU-bound tasks as NLSVMs. We measure the execution time of two MPI communications primitives from IMB suite: *Sendrecv* and *Alltoall*.

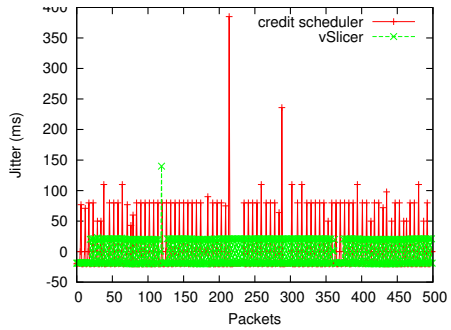


Figure 2.18.: Single line VoIP downstream jitter

In the IMB *Alltoall* benchmark, each MPI process sends a distinct message to each process in the system. A process executing this communication pattern usually sends messages to all other processes using non-blocking *sends* and waits for the receipt messages from all other processes. When vSlicer is used, each LSVM gets scheduled frequently for a micro time slice period (of 5ms), leading to more timely processing of send/receive messages to/from other processes and hence faster process of the entire MPI job. Figure 2.15 shows that, under various message sizes, vSlicer reduces the execution time by half or more, compared with the credit scheduler.

In the IMB *Sendrecv* benchmark, the MPI processes form a periodic communication chain. Each process sends a message to its right neighbor in the chain and receives a message from its left neighbor. Figure 2.16 shows the results for this benchmark. vSlicer leads to significant reduction in the execution time (up to 4.5 $\times$ improvement when the message size is 1024KB). The reduction is even higher than in the *Alltoall* case. The main reason lies in the chain of dependencies imposed by this particular communication pattern. Each process depends on its left neighbor to receive and acknowledge the message being sent; and each process depends on its right neighbor to send a complete message. The longer message processing delays incurred by the credit scheduler causes the entire messaging chain to take longer time in a *cascading* way. When vSlicer is used, each LSVM has multiple opportunities in one scheduling round to process those incoming/outgoing messages, leading to faster progress of messaging chain.

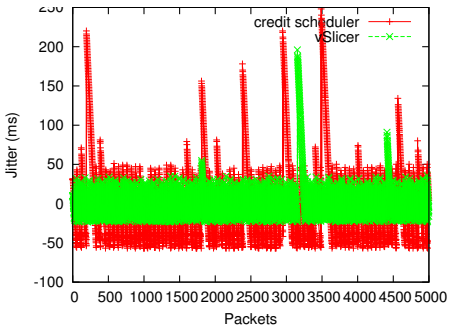


Figure 2.19.: Multi-line VoIP upstream jitter

Experiments with MyConnection Server Finally, we evaluate the effectiveness of vSlicer with latency-sensitive, soft real-time applications such as streaming media servers and VoIP gateways. We use MyConnection Server (MCS) [30] as our benchmark application.

MCS is a suite of benchmarks for assessing the performance and quality of networking and computing infrastructures for hosting soft real-time applications such as VoIP, video streaming, IPTV, and video conferencing. We use the *VoIP* test and the *streaming video* test of MCS for our evaluation of vSlicer. We run MCS in a VM which shares the same CPU core with 3 other non-idle VMs. Two of these VMs are LSVMs, including the VM that runs the MCS tests. The VoIP/media streaming clients run in another physical machine in the same LAN, but we simulate remote clients in the real world by introducing a random delay between 20ms to 40ms using the Linux *netem* module.

The VoIP test generates voice traffic of one or more active VoIP sessions with a selected audio compression algorithm. In this test, a VoIP client connects to the MCS via the SIP protocol, emulates one or more voice conversations using G.711 codec, and measures QoS metrics such as jitter, packet loss, and the discarded packet percentage.

Figure 2.17 and Figure 2.18 show the upstream and downstream jitter for the *single line* VoIP test (*i.e.* when only one VoIP session is active), respectively. Figure 2.19 shows the upstream jitter for the *multi-line* VoIP test (*i.e.* when multiple VoIP sessions are active simultaneously). Table 2.1 and Table 2.2 summarize the results of the VoIP test. Compared with the credit scheduler, vSlicer achieves a 16.6ms (62%) reduction in upstream jitter and

11ms (43%) reduction in downstream jitter in the single line VoIP test. In the case of multi-line VoIP test, vSlicer achieves a 23.7ms (65%) reduction in downstream jitter and 29.2% reduction in downstream packet loss. Under the credit scheduler, we could not even obtain meaningful downstream jitter results for the multi-line VoIP test, due to the heavy packet loss.

Table 2.1.: Single line VoIP test results under credit scheduler and vSlicer

Scheduler	Upstream Jitter	Downstream Jitter	Packets Discard
Credit scheduler	26.7ms	25.8ms	1.2%
vSlicer	10.1ms	14.8ms	0%

Table 2.2.: Multi-line VoIP test results under credit scheduler and vSlicer

Scheduler	Upstream Jitter	Downstream Packet Loss	Packets Discarded
Credit scheduler	36.7ms	44.5%	6.0%
vSlicer	13.0ms	15.3%	1.5%

Table 2.3.: Streaming video test results under credit scheduler and vSlicer

Scheduler	Video Jitter (ms)	Audio Jitter (ms)	Trip Time (ms)	SETUP Time (ms)	DESCRIBE Time (ms)	PLAY Time (ms)
Credit	46.2	41.2	110	361	480	509
vSlicer	16.6	15.8	51	176	262	243

The *streaming video* test involves video streaming sessions from the MCS to the clients via TCP based on the Real Time Streaming Protocol (RTSP) [31]. The streaming video server sends a series of audio and video packets at a fixed rate to the client. The client will

measure the packet jitter and the server will measure the trip time, which is the application-level round-trip time. The test also measures the time to perform different RTSP commands such as SETUP, DESCRIBE, and PLAY. In this experiment, the payload of each audio packet is 32 bytes and the payload of each video packet is 160 bytes. The media transmission rate is 20 packets per second (for both audio and video packets). Table 2.3 shows the results of the test. Compared with the credit scheduler, vSlicer reduces the video jitter by 29.6ms (64%) and reduces the audio jitter by 25.4ms (62%). Furthermore, vSlicer achieves significant improvements (time reduction) for all the other streaming video metrics measured.

3 ACCELERATING VIRTUAL MACHINE I/O PROCESSING USING DESIGNATED TURBO-SLICED CORE

In a virtual machine (VM) consolidation environment, it has been observed that CPU sharing among multiple VMs will lead to I/O processing latency because of the CPU access latency experienced by each VM. In the previous chapter, we presented vSlicer, a low latency VM scheduler, to reduce the CPU access delay and improve the responsiveness of I/O intensive applications running in VMs. However, we found this method can not improve the network (TCP & UDP) throughput in the datacenter, because it can only reduce the VM scheduling latency to a certain value but this value is not small enough. To get the full speed bandwith in LAN, the VM scheduling delay should be sub-millisecond level which is the maximum latency in physical host. To solve this problem, in this chapter, we present vTurbo, a system that accelerates I/O processing for VMs by offloading I/O processing to a designated core. More specifically, the designated core – called turbo core – runs with a much smaller time slice (e.g., 0.1ms) than the cores shared by production VMs. Most of the I/O IRQs for the production VMs will be delegated to the turbo core for more timely processing, hence accelerating the I/O processing for the production VMs. Our experiments show that vTurbo significantly improves the VMs' network and disk I/O throughput, which consequently translates into application-level performance improvement.

3.1 Introduction

IRQ processing delay can affect both network and disk I/O performance significantly. For example, in the case of TCP, incoming packets are staged in the shared memory between the hypervisor (or privileged domain) and the guest OS, which delays the ACK generation and can result in significant throughput degradation. For UDP flows, there is no such time-sensitive ACK generation that governs the throughput. However, since there

is limited buffer space in the shared memory (ring buffer) between the guest OS and the hypervisor, it may fill up leading to packet loss. IRQ processing delay can also impact disk write performance. Applications often just write to memory buffers and return. The kernel threads handling disk I/O will flush the data in memory to the disk in the background. As soon as one block write is done, the IRQ handler will schedule the next write and so on. If the IRQ processing is delayed, write throughput will be significantly reduced. If the OS were running directly on a physical machine, or if there were a dedicated CPU for a given VM, the IRQ processing component gets scheduled almost instantaneously by preempting the currently running process. However, for a VM that shares CPU with other VMs, the IRQ processing may be significantly delayed because the VM may not be running when the I/O event (e.g., network packet arrival) occurs.

Unfortunately, none of the existing efforts explicitly tackles this problem. Instead, they propose indirect approaches that moderately shorten IRQ processing latency hence achieving only modest improvement. Further, because of the specific design choices made in those approaches, the IRQ processing latency cannot be fundamentally eliminated (i.e., made negligible) by any of the designs, meaning that they cannot achieve close-to optimal performance. For instance, the vSlicer approach [32] schedules I/O-intensive VMs more frequently using smaller micro-time-slices, which implicitly lowers the IRQ processing latency, but not significantly. Also it does not work under all scenarios. For example, if two I/O latency-sensitive VMs and two non-latency-sensitive VMs share the same pCPU, the worst-case IRQ processing latency will be about 30ms, which is still non-trivial, even though it is better than without vSlicer (which would be 90ms). Similarly, another approach called vBalance [33] proposes routing the IRQ to the vCPU that is scheduled for the corresponding VM. This may work well for SMP VMs that have more than one vCPU, but will not improve performance for single vCPU VMs. Even in the SMP case, it improves the chances that at least one vCPU is scheduled; but fundamentally it does not eliminate IRQ processing latency because each vCPU is contending for the physical CPU independently.

To solve this problem more fundamentally, we aim to make the IRQ processing latency for a CPU-sharing VM almost similar to the scenario where the VM is given a dedicated

core. To achieve this, we propose a new solution called vTurbo, that involves two basic ideas. First, we leverage the existence of multiple cores in modern processors to designate a specialized turbo-sliced core (or *turbo core* for short), for synchronous processing threads in the guest OS. In terms of actual hardware, the turbo core is no different from a regular core, except that the hypervisor-level scheduler schedules VMs on this core with *extremely small quantum* (e.g., 0.1ms). Second, we expose this turbo-sliced core to each VM as a "co-processor" just dedicated to kernel threads that require synchronous processing, such as IRQ handling. The other regular kernel threads are scheduled on a regular core with regular slicing just like what exists today. Since the IRQ handlers are executed by the turbo core, they are handled almost synchronously with a magnitude smaller latency. For example, assuming 5 VMs and 0.1ms quantum for the turbo core, an IRQ request is processed within 0.5ms compared to 150 ms (assuming 30ms time slice for regular cores).

The turbo core is accessible to all VMs in the system. If a VM runs only CPU-bound processes, it may choose not to use this core since its performance is not likely to be good due to frequent context switches. Even if a VM chooses to schedule a CPU-bound process on the turbo core, it has virtually no impact on other VMs' turbo core access latency thus providing good isolation between VMs. We ensure fair-sharing among VMs with differential requirement between regular/turbo cores because, otherwise, it would motivate VMs to push more processing to the turbo core. Thus, for example, if there are two VMs—VM1 requesting 100% of the regular core, and VM2 requesting 50% regular and 50% turbo cores, the regular core will be split 75-25% while VM2 obtains the full 50% of the turbo core, thus equalizing the total CPU usage for both VMs. We also note that, while we mention one turbo core in the system, our design seamlessly allows multiple turbo cores in the system driven by I/O processing load of all VMs in the host. This makes our design extensible to higher bandwidth networks (10Gbps and beyond) and higher disk I/O bandwidths that require significant IRQ processing beyond what a single core can provide.

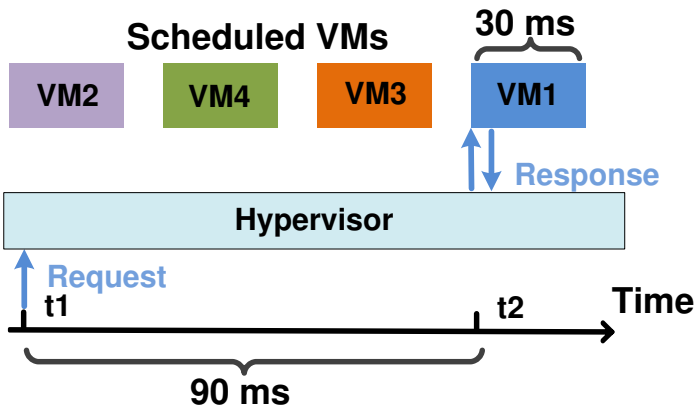
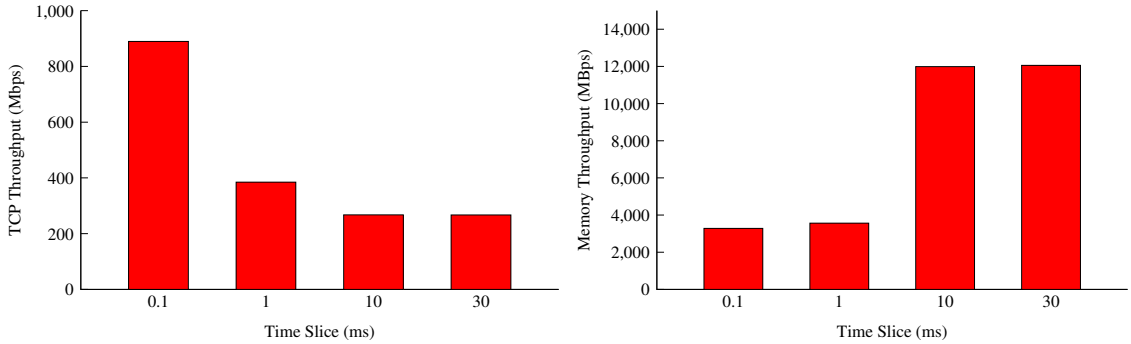


Figure 3.1.: Impact of VM CPU sharing on I/O processing.

3.2 Motivation

Let us first focus on receive-side I/O processing. In a non-virtualized system, all receive-side I/O events (*e.g.*, network packet arrival) are typically handled by specific IRQ routines corresponding to each device (*i.e.*, disk controller or NIC) in the OS kernel. The data is stored in a kernel buffer first, and once the user process is scheduled, it copies the data from the kernel buffer to the user buffer. Since I/O-bound processes usually have higher priority, they get scheduled relatively quickly and the data is subsequently processed by the application thus achieving high I/O throughput. However, in a virtualized system with several VMs sharing a physical CPU, each VM gets only a slice of the physical CPU, which means the incoming I/O event will need to wait until the VM gets access to the CPU. Such a CPU access latency will significantly affect the timeliness of IRQ processing, resulting in low I/O throughput.

We illustrate this negative effect using an example shown in Figure 3.1. In this example, 4 VMs share a physical CPU. VM1 runs a mixed workload that includes both CPU-bound tasks and I/O-bound applications, while VM2 to VM4 run only CPU-bound applications. Assuming a proportional-share VM scheduling policy (adopted by Xen and VMware ESX), VM1 gets only 25% of CPU when all VMs are busy, which means that roughly 75% of time, VM1 has to wait in *runqueue* and cannot process I/O events immediately. When an I/O request for VM1 reaches the hypervisor at t_1, VM1 cannot process this request and



(a) TCP throughput (b) Memory throughput

Figure 3.2.: Impact of micro-timeslice on TCP throughput and memory throughput

respond until t_2. If the I/O-bound application in VM1 is a TCP server, for instance, the client will stop sending data to the server once the client's TCP window is full, due to lack of acknowledgments from the server while VM1 is in *runqueue*. If VM1 runs a UDP server, even though the client can continue to send data to the server without getting responses, the packets will be dropped by the hypervisor once the shared buffers (between the hypervisor and guest OS) are full. As a result, throughput of either TCP or UDP for VM1 would be much lower than the available capacity.

In the reverse direction (*i.e.*, when a process sends packets or writes to the disk), the user process first copies data to the kernel buffer associated with the particular output (*e.g.*, socket, file descriptor). For some I/O mechanisms such as asynchronous network packet sends and disk writes, the call to output the data will return to the user process immediately after the data is copied to the kernel buffer. The kernel components associated with the corresponding device will asynchronously write the data to the device. However, this task cannot be continued efficiently if the hypervisor schedules the vCPU out while the kernel component is waiting for the completion of the write to the device, resulting in low throughput.There are other sources of delay for interrupt processing even after the I/O event reaches the VM. These include long periods in which, the VM runs with interrupts disabled, locking conflicts for shared data structures (such as TCP accept queue [34]) and overhead of dispatching interrupts in virtualized environments [35]. However, most of these latencies

lie within sub-millisecond range in the average case [36, 37], while the scheduling delay causes the interrupt processing to be delayed for tens of milliseconds (in our example, the average scheduling delay is about 35ms for Xen VMM).

Symmetric multi-processing (SMP) VMs can take advantage of a multi-core architecture to execute many different applications in parallel and improve the overall system throughput. In an SMP-VM, two or more allocated vCPUs are scheduled by the hypervisor scheduler on any available pCPUs and thus, each vCPU has a higher chance to get scheduled. However, the SMP-VM may still suffer from scheduling delays, if none of the vCPUs can be scheduled in because the pCPUs are all busy executing other vCPUs.

Thus, we cannot guarantee that the vCPU running an IRQ gets scheduled in time when a target VM receives an I/O request.

3.2.1 Existing Approaches

Now we discuss several existing approaches addressing the problem of CPU sharing impacting I/O performance of VMs and discuss why they do not work well.

Reducing CPU time-slice. One intuitive approach to solve the scheduling latency problem is to uniformly reduce the VM scheduling time-slice [18]. In proportional-share scheduling, the worst-case scheduling delay of each VM is *(Number of sharing VMs -1)* $\times$ *time-slice*. A small scheduling time-slice enables VMs to get scheduled more frequently thus improving the I/O throughput of VMs. However, the short time-slice results in more frequent context switches which may hurt the performance of memory-intensive or CPU-bound applications. We conduct a simple experiment to demonstrate this problem.

In our experiment, 4 single vCPU VMs share one physical CPU. One VM hosts a TCP server, the client is running in another physical machine in the same LAN. Iperf [26] is used to measure the server's TCP throughput. We vary the scheduling time-slice from 0.1ms to 30ms, which is the default time-slice of Xen. From Figure 3.2(a) we can find that, smaller time-slice leads to higher TCP throughput. Especially, with a 0.1ms time-slice, the average TCP throughput is up to 900 Mbps which is close to the bandwidth of

1Gbps network card used in our experiment. However, the performance of memory/CPU bound applications degrades under smaller time-slice as shown in Figure 3.2(b). Here, we run STREAM [23] benchmark in one of the 4 VMs[1]. So, simply reducing the CPU time-slice cannot simultaneously benefit both I/O-intensive applications and CPU-intensive applications. Hence this approach is not suitable for cloud environments where mixed workloads are common.

Sending I/O interrupts to active vCPU To reduce the IRQ processing delay and improve I/O throughput for SMP-VMs, a recent approach called vBalance [33] sends I/O interrupts to the active vCPU of the target VM. In this way, I/O interrupts can be processed in a more timely fashion and I/O throughput may be improved. However, there are still several issues with this method. As discussed before, an SMP-VM may have increased chances to get scheduled because of the multiple vCPUs assigned to it. But there is no fundamental guarantee that the SMP-VM have at least one vCPU running at any time. If none of the vCPUs is running, an I/O interrupt still cannot be processed in time. Besides, even if the I/O interrupt is sent to an active vCPU successfully, the I/O cannot be finished if the vCPU executing the I/O application is not running simultaneously. This specifically impacts TCP, where the application vCPU may be in the *runqueue* holding the ownership of the lock structure, hence the kernel-level TCP processing cannot generate an ACK in time for incoming TCP packets. We suspect this is the main reason [33] only reports 400Mbps TCP throughput in a 1Gbps LAN environment.

Differentiated VM scheduling Tuning VM scheduling policy is another method to speed up I/O processing. vSlicer [32] schedules each latency-sensitive VM (LSVM) more frequently with a smaller micro time-slice, which enables more timely processing of I/O events by LSVMs. There are two caveats of this approach. First, we need to know which VMs are LSVMs running latency-sensitive applications in advance and adjust the VM scheduler configuration accordingly.

[1]We conducted a similar experiment in [32]. But here we set even smaller time-slice (0.1ms) and contrast TCP and memory throughput under such a time-slice.

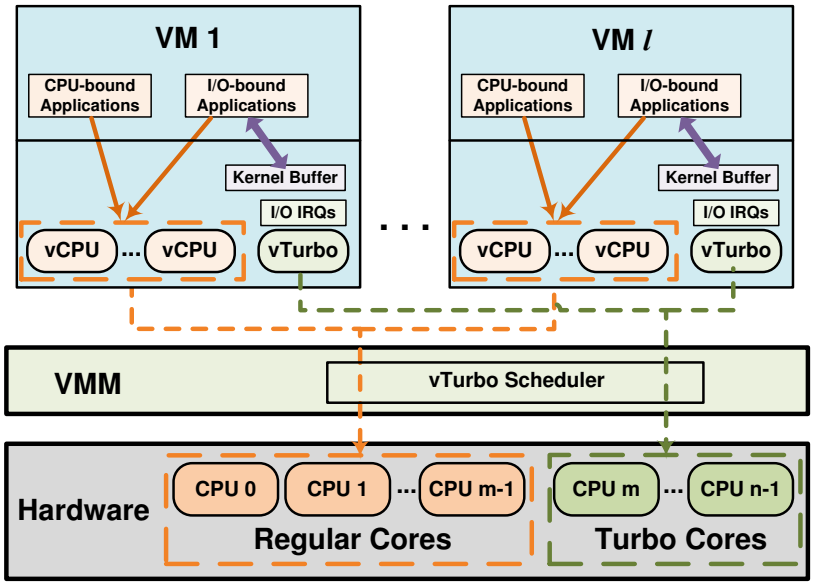


Figure 3.3.: Architecture of vTurbo

Second, vSlicer reduces the scheduling delay but does not completely eliminate it, as discussed earlier. It, therefore, does not improve the TCP/UDP throughput significantly, although it does reduce application-perceived I/O latency.

3.3 Design

The discussion in the previous section suggests that if we use a very small value as the CPU time-slice, I/O performance of CPU-sharing VMs can be significantly improved. However, we also showed that such an approach may hurt the performance of CPU-bound VMs, for which larger time-slice is desirable. To address this dilemma, we leverage one *key degree of freedom* that has not been exploited hitherto: The CPU time-slice for each core may *not* be the same for a multi-cores system.

Thus, in our approach called vTurbo, we designate one (or more) core(s) in the system as what we call a *turbo core*, which is just any regular physical core, except that we set a very small (*e.g.*, 0.1ms) CPU time-slice for it. We expose the turbo core to each VM *in*

addition to the regular cores, and allow the guest OS to schedule I/O-bound threads (*e.g.*, IRQ handling) in the turbo core thus speeding up I/O processing significantly.

The guest OS still schedules CPU-bound workloads on cores with the regular time-slice. As such vTurbo achieves I/O processing speedup without impacting CPU-bound workloads.

In effect, vTurbo focuses on re-factoring the interface between the hypervisor and guest OS, with the new abstraction of turbo core. This approach is completely transparent to applications running in VMs, a key advantage of practicality. Another benefit of vTurbo is that it does not require classification of VMs into I/O- or CPU-intensive VMs, as required by some solutions such as vSlicer [32]. Such classification is difficult as most VMs in practice run a combination of I/O and CPU workloads. Of course, the guest OS now needs to identify I/O-bound threads such as IRQ processing and schedule them on the turbo core. But that is not hard as there are only a handful of such threads. Our approach also guarantees CPU *fairness* among all VMs. Any VM using the turbo core will essentially not obtain any "extra" CPU beyond its fair share—an important property in multi-tenancy clouds.

The architecture of vTurbo requires changes to both the hypervisor and the guest OS. At the hypervisor level, the VM scheduler needs to accommodate the new turbo core abstraction. At the guest kernel-level, we need to modify the VM process scheduler to pin certain threads to the turbo core in addition to a few changes to the TCP protocol stack. In the following subsections, we discuss these in more detail.

3.3.1 Modifications to Hypervisor

We mainly need to modify the VM scheduler in the hypervisor to support the turbo core abstraction.

Upon host initialization, we designate a set of cores in the host as turbo cores. The number of turbo cores is configurable, and our current version statically assign turbo cores based on user configuration. However, we believe that our system can be improved by having a dynamic method to assign turbo cores based on the available machine capacity

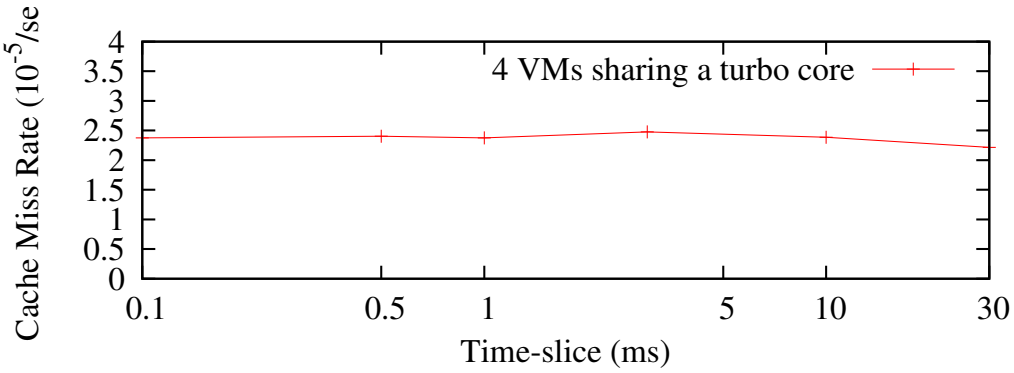


Figure 3.4.: Impact of time-slice size on cache misses on turbo cores.

(*i.e.*, total number of cores), number of VMs, demand for the turbo core, and overall I/O intensity (*e.g.*, a host with multiple active NICs or 10GB/s NICs may require more turbo cores). One can also dynamically change the number of turbo cores via administrative tools (such as *xm* tools in Xen). While the current implementation of vTurbo randomly selects the turbo cores, we can incorporate parameters such as cache affinity to further improve their performance.

In vTurbo, each VM is assigned a *turbo vCPU* in addition to its regular vCPUs. The turbo vCPU is assigned to one of the turbo cores in the host. This step is performed during VM initialization. For instance, if a user launches an SMP-VM configured with 2 vCPUs, the VM will have 3 vCPUs after initialization. Among these, the 0^{th} vCPU is the turbo vCPU, whereas the 1^{st} and 2^{nd} vCPUs are regular vCPUs.

Based on our empirical study (discussed in motivation section), we set 0.1ms as the CPU scheduling time-slice for turbo cores (as it enables the VM to reach up to 900Mbps TCP throughput for a 4 CPU-sharing VMs scenario). Since only interrupt processing runs on turbo cores, frequent context switches caused by the small turbo core time-slice does not affect the performance of interrupt processing much because of the very short duration of the processing. According to our measurements (Figure 3.4), when 4 VMs each running an iperf server share one turbo core, the order of magnitude of cache miss per second on the turbo core is only 10^{-5}, which is negligible. The CPU time-slice for regular cores is set be a much larger value — 30ms in the current implementation which is the default time-slice of

Xen. The vTurbo VM scheduler uses per-core scheduling timer to trigger scheduling code to select the next vCPU from the *runqueue*. We achieve CPU time-slice differentiation by setting these timers to 0.1ms for turbo cores and 30ms for regular cores.

Once the vCPUs are assigned to turbo cores and regular cores, our next concern is to correctly handle vCPU migration in the presence of turbo cores. vCPU migration allows to balance the CPU load among the available cores in the system. However, if we let the vCPUs to migrate freely among available cores, there is a possibility that a regular vCPU be migrated to a turbo core making undesirable effects. To solve this, we restrict migration of regular vCPUs to only among all regular cores and migration of turbo vCPUs only among turbo cores. We do not allow a turbo vCPU to migrate to a regular core or vice versa. This is done by changing each vCPU's affinity to the corresponding set of cores. Hence vTurbo scheduler not only determines the appropriate mapping between vCPUs and physical cores, but also ensures fair CPU sharing among all VMs.

VM scheduling policy Since we intend to use the turbo core only for I/O activity, we cannot treat it as a regular core and apply the existing scheduling policy to guarantee fair sharing among VMs. The challenge is to determine the CPU share of VMs for turbo and regular cores in the presence of heterogeneous workloads (*i.e.*, when a VM is CPU intensive, I/O-intensive, or both).

Current schedulers (*e.g.*, Xen's) use simple credit-based scheduling algorithm for achieving global load balancing and work-conservation. For instance, in Xen's credit scheduler, a VM is assigned some amount of credits periodically based on the priority of the VM. As the vCPUs belonging to a particular VM run on physical CPUs, credits are deducted from that VM. When the scheduler needs to make a decision, it uses the amount of available credits for each VM to decide which vCPU will run on the physical CPU. To accommodate turbo cores in our system, we mainly need to modify the credit assignment portion of the credit scheduling algorithm to account for the turbo vCPU execution time.

Specifically, assume l VMs are sharing an n-core host with m regular cores and n-m turbo cores. Let rd_i denote the percentage demand for regular cores (CPU-bound component) and let td_i denote the percentage demand for turbo cores (I/O-bound component) for

VM_i. We assume the demand for regular core and turbo core in two consecutive scheduling periods does not change much (if it does, we account for and adjust it in future rounds). So both rd_i and td_i are calculated based on the consumed CPU cycles by the VM in the previous scheduling period. Since our scheduler is work-conserving, the division of the total capacity among the regular and turbo cores is determined by the following:

$$C^R_{tot} = \sum_{i=1}^{l} rd_i \ \ and \ \ C^T_{tot} = \sum_{i=1}^{l} td_i$$

The total capacity demand of the system is:

$$C_{tot} = C^R_{tot} + C^T_{tot}$$

The fraction of CPU allocated for a VM out of this total capacity is determined by its assigned weight wt_i. Hence each VM's fair share (FS_i) of CPU is given by:

$$FS_i = (C_{tot} \times wt_i)/(\sum_{j=1}^{l} wt_j)$$

In vTurbo, we first allocate turbo core capacity fairly among VMs, as all of the VMs' IRQ processing is performed by the turbo vCPUs and starvation of turbo vCPUs (even for CPU-bound VMs) will result in application performance hit. So VM_i's fair share of the turbo core (FS^T_i) is calculated as:

$$FS^T_i = (C^T_{tot} \times wt_i)/(\sum_{j=1}^{l} wt_j)$$

Once VM_i's turbo core share is determined, we allocate the rest of its CPU share from the regular cores. The fraction of the allocation is given by:

$$FS^R_i = FS_i - \hat{FS^T_i}$$

where $\hat{FS^T_i}$ denotes the *actual* usage of the turbo core by VM_i in the previous scheduling period. We use FS^T_i and FS^R_i to determine the proportion of credits given to VMs out of total credits in the turbo core pool and regular core pool, for the next scheduling period. Table 3.1 shows the CPU allocation results from experiments with our prototype, where two VMs—with equal weight—share one regular core and one turbo core, under various

workload demands. Columns 2 and 3 of the table indicate the CPU demand of each VM (*i.e.*, CPU utilization if they were run without CPU sharing); Columns 4 and 5 indicate measured consumption in the previous scheduling period; Columns 6 and 7 indicate the allocated shares of regular and turbo cores based on our policy; and Columns 8 and 9 show the *measured* consumption of regular ($\hat{FS}_i^R$) and turbo ($\hat{FS}_i^T$) core capacity in the next scheduling period. The results confirm that our policy allocates CPU with proportional fairness.

Table 3.1.: VMs' CPU demand and allocated CPU shares under different scenarios

	Demand		Measured		Allocated		Consumed	
	Reg.	Turbo	rd_i	td_i	FS_i^R	FS_i^T	$\hat{FS}_i^R$	$\hat{FS}_i^T$
VM1	100	0	50	0	50	0	50	0
VM2	100	0	50	0	50	0	50	0
VM1	100	0	50	0	100	0	100	0
VM2	100	100	50	100	0	100	0	100
VM1	100	100	50	50	50	50	50	50
VM2	100	100	50	50	50	50	50	50
VM1	100	15	50	15	70	35	70	15
VM2	100	55	50	55	30	35	30	55

3.3.2 Modifications to Guest OS

Process scheduler As noted before, if CPU-bound workload were scheduled on the turbo cores, its performance would degrade due to frequent context switches. Since process scheduling inside the VM is transparent to the hypervisor's VM scheduler, we should make the guest OS's process scheduler aware of the turbo core to prevent user processes and non-I/O-related kernel threads from being scheduled on the turbo core. This can be achieved by setting scheduler affinity rule which sets the affinity of the non-I/O related threads to regular vCPUs. In Linux, this can be easily done by a scheduling mechanism known as Linux CPU isolation [38].

I/O buffers in guest OS With the above change, we can reduce IRQ processing delay to extremely small values. However, low IRQ processing delay by itself does not automatically translate into high I/O throughput, because of a critical *locking behavior* between the kernel and application threads as we explain below. The network receive path in typical OSes (*e.g.*, Linux) consists of two main steps: (1) Processing IRQ in kernel and buffering data in kernel buffer; (2) Application reading the data from kernel buffer and clearing it. Since the CPU time-slice of regular cores is still 30ms in vTurbo, the CPU access delay on the regular core will make the kernel buffer full very quickly and stop the IRQ threads from buffering more data, which would lead to poor I/O performance.

To address this problem and to keep the turbo vCPU busy processing IRQs, we need to tune the kernel buffer to store more received data while the application running on regular vCPU is blocked. As an example when 4 single-vCPU (excluding turbo vCPU) VMs are sharing one regular core, the CPU access delay is up to 90ms (*(4 - 1)* $\times$ *30ms*). To keep the IRQ threads on turbo vCPU busy, all data received during this period need to be buffered. So if the bandwidth of NIC is B_N, the minimum kernel buffer required (B_{min}) is: $B_{min} = B_N \times Scheduling\_Delay$ (*i.e.*, the required kernel buffer is proportional to the number of VMs sharing the same CPU core). In fact, the real kernel buffer we need is almost always much larger than B_{min}. For example, in our experimental environment with 1Gbps NICs, if 4 VMs share one CPU, the kernel buffer for UDP should be around 11.25MB. However, we did not obtain high throughput (more than 900Mbps) until we set the UDP kernel buffer (*net.core.rmem_max*) to about 40MB.

Modifications to VM's TCP stack While simply setting the guest kernel buffer to a high value ensured good UDP performance, it did not improve TCP throughput at all. Upon a deeper investigation, we found the following problem: In TCP, when a data segment is received, the receiver generates an ACK to inform the reception of the segment. The sender uses this ACK to confirm the reception of data as well as for congestion control.

Now, using the turbo core, we eliminate the long delay for processing incoming data segments. With our additional I/O buffering enabled, the IRQ context now buffers all these

Algorithm 2 Generating ACK for Backlog Queue

```
1: rcv.nxt is the seq. number of expected packet for receive queue
2: bl.nxt is the seq. number of expected packet for backlog queue
3: seq is the seq. number of received packet
4: if backlog_queue is empty then
5:     if rcv.nxt ≥ bl.nxt then
6:         /* initial status or packets in backlog queue are all acked by process context */
7:         bl.nxt = rcv.nxt;
8:         bl.online = 1; /* enable ACK generation */
9:     end if
10: else
11:     if bl.online == 0 and bl.nxt ≤ rcv_nxt then
12:         /* packets in backlog queue are acked by process context */
13:         bl.online = 1; /* enable ACK generation */
14:         bl.nxt = rcv.nxt;
15:     end if
16: end if
17: if bl.online == 1 then
18:     if bl.nxt == seq then
19:         /* packet to be added to backlog queue is in order */
20:         update(bl.nxt);
21:     else
22:         /* stop ACK generation due to out-of-order packet */
23:         bl.online = 0;
24:     end if
25: end if
26: if add_backlog() is successful and bl.online == 1 then
27:     tcp_ack_backlog(); /* generate and send ACK */
28: end if
```

data packets. However, the locking behavior in the VM's TCP stack still prevents the ACK generation in a timely manner, hence reducing TCP throughput significantly.

Specifically, when the user process is calling function *recv()*, it locks the socket to prevent the IRQ threads from modifying the socket structure while it is reading from the socket buffers. If a new data segment arrives during this period, the IRQ process will queue it in the *backlog* queue without generating an ACK. When the receiving process engages in a tight receiving loop, the socket gets locked frequently by the process context. Moreover, the process can get scheduled out of the regular core while it is holding the lock. When this happens, ACKs will not be generated for a long period (until the process gets scheduled and releases the lock), even though the turbo core can accept and buffer TCP segments from the network. As a result, the sender will throttle down the sending rate leading to sub-par TCP throughput.

We make a simple modification to the VM's TCP stack to enable ACK generation from the IRQ context running in the turbo core, even when the socket structure is locked by the user process. The high-level steps performed by our modification are shown in Algorithm 2 which runs in the *softIRQ* context just before queuing the packet in the TCP backlog queue. Here, when the IRQ thread discovers that the socket is locked by the user process, it checks whether the new data segment is in-order. If so, an ACK is generated for the data packet, which will then be marked as acknowledged and queued. Note that we are not modifying the socket structure as it is currently owned by the process context. This is somewhat similar to vSnoop [5], although vSnoop is implemented purely in the driver domain whereas the ACK generation here is from within the guest VM. Thus we have access to VM's TCP information and can afford much larger buffers (compared to the limited ring buffer space in vSnoop). If a flow encounters an out-of-order packet, we disable this ACK generation until the missing segments are recovered by the usual slow path of TCP processing. This small modification helps achieve TCP throughput close to the line rate.

3.4 Implementation

We have implemented a prototype of vTurbo based on Xen 4.1.2. vTurbo only requires small modifications to the VM scheduler in hypervisor (about 400 lines of code) and guest OS kernel (less than 200 lines of code).

Hypervisor To differentiate between regular cores and turbo cores, we added a field to the per-core data structure *schedule_data*, to indicate the CPU time-slice for the specific core–30ms for regular cores and 0.1ms for turbo cores. Our implementation allows the flexibility of changing these values dynamically via *xm* tools.

Our vTurbo scheduler inherits most of its functionality from Xen's credit scheduler which provides the proportional fairness and work-conserving properties. We added and modified functionality of the main scheduler code of the credit scheduler to accommodate turbo cores and turbo vCPUs. Specifically, we modified function *csched_schedule()*, which is responsible for selecting vCPUs from the *runqueue* to run on physical cores and setting the scheduling timer of turbo cores to 0.1ms.

We assign each VM a turbo vCPU by modifying the VM's configuration so that an extra vCPU is added during the configuration parsing step of VM initialization performed by the Xen tools. Also during this step, the turbo vCPUs are pinned to the set of turbo cores and regular vCPUs are pinned to the regular cores by modifying the loaded VM's configuration. By doing this, we do not have to modify the scheduler code to prevent undesirable vCPU migrations (discussed in Design section), because the credit scheduler will adhere to the CPU affinity rules set in the configuration.

CPU accounting is conducted by function *csched_acct()* in the credit scheduler. We extended this function by implementing two accounting routines for regular and turbo vCPUs individually as shown in Algorithm 3. They run at different frequencies in accordance with CPU scheduling frequencies (e.g., 30ms for regular vCPUs and 0.1ms for turbo vCPUs), because updating credits faster or slower than the scheduling frequency would cause inaccurate state of vCPUs in terms of *OVER* and *UNDER* priorities in Xen. The vTurbo accounting routines are simple, incurring very low overhead considering the high

Algorithm 3 vTurbo accounting algorithm

Require: $num\_tcore \geq 1$

Require: $num\_rcore \geq 1$

Require: $num\_vm \geq 1$

Regular_accounting triggered every 30ms:

$tcore\_usage = get\_rcore\_usage()$; /* C_{tot}^{R} */

$rcore\_usage = get\_tcore\_usage()$; /* C_{tot}^{T} */

for *vm* in *vm_list* **do**

$vm.credits = vm.weight \times$

$(tcore\_usage + rcore\_usage)/vm\_weight\_sum$;

$vm.tcredits = get\_turbo\_core\_usage(vm)$;

$vm.rcredits = vm.credits - vm.tcredits$;

$ratio = 300$; /* = 30/0.1 */

$vm.vturbo\_slice = vm\_vcredits/ratio$;

$update\_rcredits(vm.rcredits)$;

end for

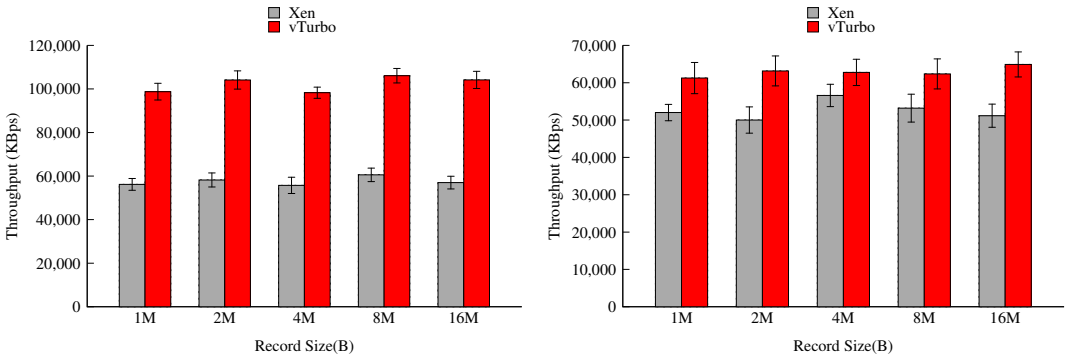
vTurbo_accounting triggered every 0.1ms:

for *vm* in *vm_list* **do**

$update\_tcredits(vm.vturbo\_slice)$;

end for

frequency of their execution. Functions *get_rcore_usage()* and *get_tcore_usage()* retrieve the consumed clock cycles by regular vCPUs and turbo vCPUs of all VMs respectively; while functions *update_rcredits()* and *update_tcredits()* set the calculated credits for regular cores and turbo cores for the next scheduling period. Function *get_turbo_core_usage()* retrieves the the clock cycle usage by the turbo vCPU of a given VM. We do not change method *burning_credits()* in the credit scheduler, which deducts credits from the VMs based on their running time on the cores. Instead we implement a new method for vTurbo credit deduction.



(a) Write (b) Read

Figure 3.5.: File read/write throughput.

Guest OS Our modification to the TCP stack, to generate early ACKs for packets buffered in backlog queue, is mainly in function *tcp_v4_rcv()*. There are 3 kernel buffers to buffer received TCP packets: (1) receive queue, (2) prequeue, and (3) backlog queue. When a socket is not locked, received packets are buffered in receive queue. However, if the application process locks the socket while fetching data from the kernel, packets received during that period will be buffered in backlog queue. We modified the backlog queuing path of function *tcp_v4_rcv()* to verify a received packet is "expected" and if so, call function *tcp_ack()* to generate an ACK for the received packet. Since very few packets (less than 0.1%) go to prequeue in CPU sharing VMs, we disable prequeue in vTurbo to simplify our implementation.

3.5 Evaluation

We first evaluate the effectiveness of vTurbo for different types of I/O operations via a series of micro-benchmarks. We then use NFS, SCP, and Apache Olio [39] to evaluate the application-level performance improvement by vTurbo.

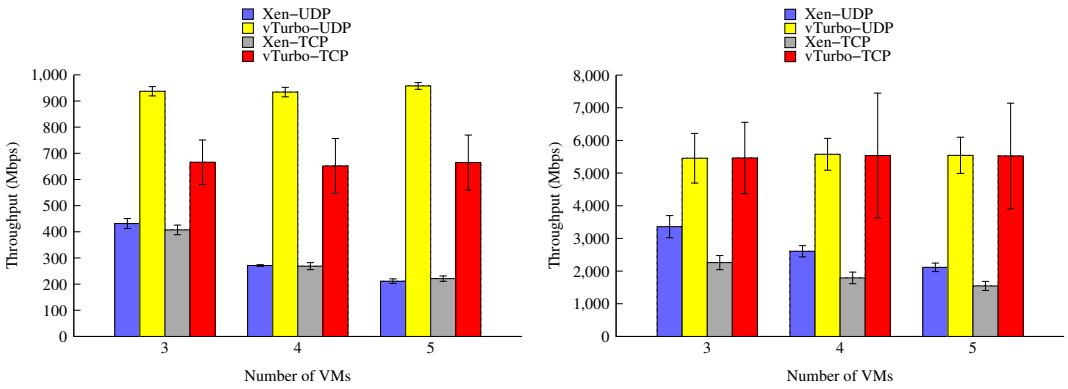
Experimental setup Our testbed consists of servers with quad-core 3.2GHz Intel Xeon CPUs and 16GB of RAM. They are connected via Gigabit Ethernet, except for the experi-

ments with 10Gbps Ethernet. These servers run Xen 4.1.2 as hypervisor and Linux 3.2 in both domain0 and guest VMs. We pin domain0 to one of the cores in all our experiments.

3.5.1 Micro-Benchmark Results

In this section we evaluate the performance of vTurbo for various types of I/O. We use *lookbusy* [25] to keep the CPU utilization at determined levels during experiments.

File read and write We use IOzone benchmark [40] to read/write a 1GB file from/to disk and measure the read/write throughput. Figure 3.5 shows the read and write throughput—in comparison with the vanilla Xen–when we vary the record size from 1MB to 16MB.



(a) 1Gbps Network (b) 10Gbps Network

Figure 3.6.: TCP and UDP throughput.

From Figure 3.5(a) we see that the disk write throughput is improved significantly (by 75% to 82%); whereas the disk read throughput (Figure 3.5(b)) sees less improvement (only up to 26%). The main reason is that, when the process performs a write, the data is immediately written to the file system cache and the *write()* call returns. So the process can keep writing while the regular vCPU is scheduled. The dirty pages of the disk cache are flushed to the disk by a kernel thread executed by the turbo vCPU. Therefore with vTurbo, disk write throughput is greatly improved. However, when the process performs a read for a fresh block from the disk, it gets blocked until the actual data blocks are read from the

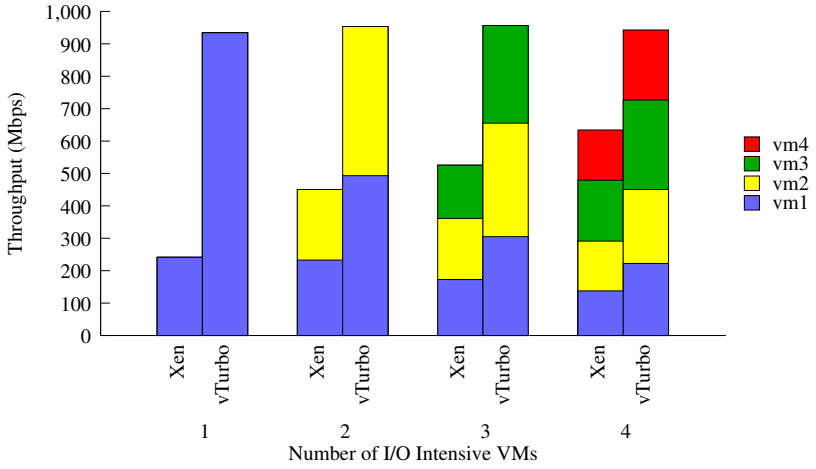


Figure 3.7.: UDP throughput: multiple I/O-intensive VMs.

disk. Meanwhile the hypervisor may schedule other vCPUs on the regular core. The turbo vCPU will be able to handle the disk read completion interrupt and place the data in the process' buffer while the regular vCPU is scheduled out. But the process will not be able to make further read requests until it is scheduled again. Hence in this case, vTurbo achieves less throughput improvement than in the case of disk write.

UDP throughput To measure the benefit of vTurbo to network I/O we first measure the UDP throughput improvement achieved by vTurbo. In these experiments, we use iperf to send a stream of UDP packets for 10 seconds to a VM sharing a core with 2, 3, or 4 other VMs. The average throughput (averaged over 10 runs) observed at the VM on vanilla Xen and vTurbo is shown in Figure 3.6(a) by blue and yellow bars, respectively. With vanilla Xen, the UDP throughput starts to decrease when the number of VMs sharing the core increases. This is because, when UDP packets arrive at domain0, the target VM may not be scheduled and the packets have to be buffered in domain0. But the space in domain0 is limited and hence once this buffer fills up, packets will be dropped causing the throughput to go down. With vTurbo, the target VM's network IRQ processing threads get scheduled frequently and hence the buffer in domain0 can be drained frequently. This leads to much less packet drops thereby achieving close-to full network bandwidth (1Gbps).

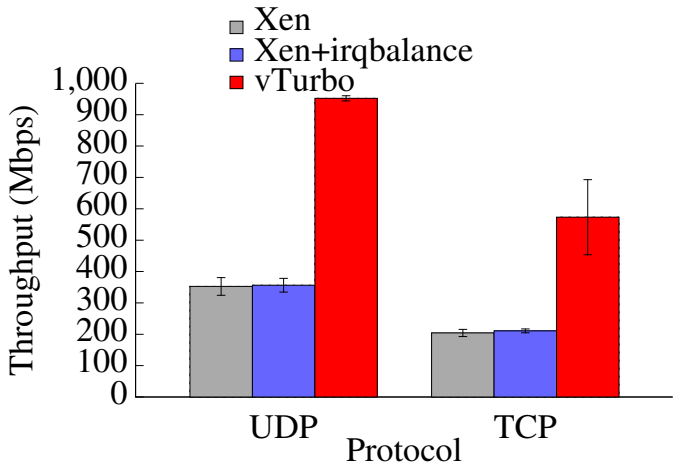


Figure 3.8.: UDP and TCP throughput for VSMP VMs.

Next, we evaluate the impact of sharing the turbo core among multiple I/O-intensive VMs. We reuse the setup in the previous experiment. But instead of 1 VM receiving a UDP packet stream, we increase the number of VMs receiving UDP streams from 1 to 4. Figure 3.7 shows the aggregate throughput achieved as well as the throughput seen by individual VMs. In both vanilla Xen and Xen with vTurbo configurations, we see that the I/O bandwidth is fairly shared among VMs. However, vTurbo achieves (close to) wire speed and outperforms vanilla Xen irrespective of the number of I/O-intensive VMs.

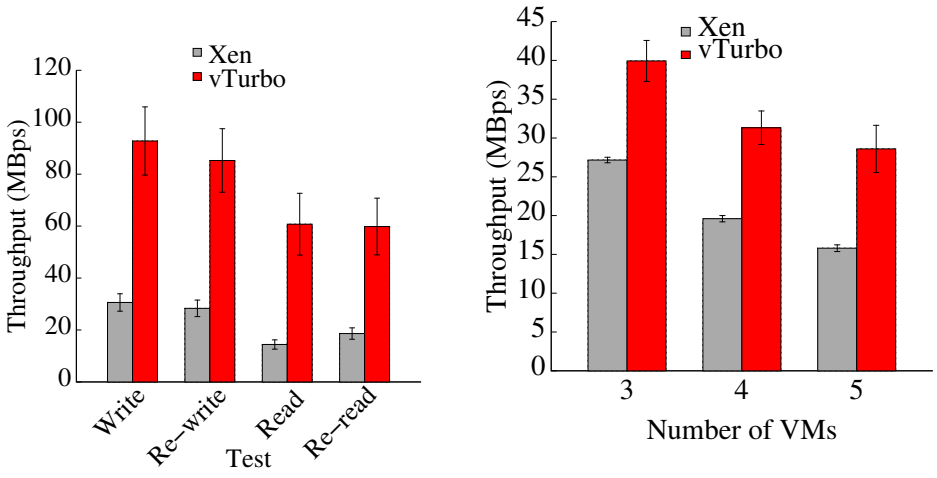
TCP throughput We use a setup similar to the UDP experiments to measure the TCP throughput improvement achieved by vTurbo. In this experiment, we send a 200MB file using iperf to a VM from another server and we vary the number of VMs sharing the same core with the receiving VM. Figure 3.6(a) shows the TCP throughput on vanilla Xen and Xen with vTurbo by grey and red bars, respectively. Recall that with vTurbo, the TCP stack is modified to generate ACKs when the regular vCPU is holding the socket ownership and scheduled out. As the figure shows, vTurbo improves TCP throughput significantly (by 63% - 200%). However the TCP throughput achieved by vTurbo still does not reach the full available network bandwidth. The reason is, even with our modification, if a packet

loss happens, we have to resort to the (usual) slow code path where packet loss recovery is subject to regular vCPU scheduling delay, which negatively affects the TCP throughput.

10Gbps Ethernet To evaluate the benefit of vTurbo with 10Gbps Ethernet, we repeat the UDP and TCP experiments. In our setup, two physical servers are connected via 10Gbps Ethernet. In the UDP experiment, we use netperf [41] to send a 10-second UDP stream to the target VM sharing a core with 2 to 4 other VMs. In the TCP experiment, we send a 500MB file using iperf from one physical server to a VM running in the other server, varying the number of VMs sharing the same core with the receiving VM. The results in Figure 3.6(b) indicate that, in a 10Gbps network, vTurbo achieves a pattern of improvement for both UDP and TCP throughput similar to that in the 1Gbps network. However, since the regular core is shared by multiple VMs, the application does not get enough CPU cycles to copy the buffered data from kernel space to user space, hence we can not achieve line speed.

Benefit of vTurbo to VSMP VMs To show the benefit of vTurbo to SMP VMs, we use iperf to send TCP and UDP traffic (in different runs) to a VM which is assigned 2 vCPUs. In this experiment, we run 4 VMs each with 2 vCPUs. These vCPUs are restricted to run in the first 2 cores of the quad-core processor, but are allowed to migrate between the two cores. Similar to previous experiments, we pin domain0 to the 3rd core and, for vTurbo, we use the 4th core as the turbo core. In the vanilla Xen configuration, we first disable *irqbalance* in VM and allow the interrupts to be directed only to vCPU0 of the VM. Next we enable *irqbalance* so that the interrupts can be balanced between the two vCPUs. In the vTurbo configuration, interrupts are routed to the turbo vCPU. Figure 3.8 shows the TCP and UDP throughput when transferring 200MB of data to the VSMP VM. vTurbo vastly outperforms both *irqbalance*-on and *irqbalance*-off configurations. However, the TCP throughput is lower than that under the "4 single-vCPU VMs" configuration (for both vanilla Xen and vTurbo configurations – see Figure 3.6(a)) . We conjecture that this is due to the vCPU migrations between the two physical cores and the iperf receiver process migrations between the two vCPUs of the VSMP VM.

3.5.2 Application-Level Results



(a) NFS read/write throughput.

(b) SCP throughput.

Figure 3.9.: SCP and NFS throughput.

NFS server throughput NFS uses TCP to transport commands and data blocks between the NFS client and server. We use the NFS tests in IOzone to evaluate the benefit of vTurbo to the NFS server. We export a directory of a VM using NFS and run IOzone in another server which mounts this exported directory. We pin the NFS server VM's vCPU to a single core shared by three other VMs with 30% CPU utilization. Figure 3.9(a) shows the file read and write throughput (file size: 1GB). vTurbo significantly outperforms vanilla Xen for all types of operations. The results for "Read" and "Re-read" operations are especially interesting (and somewhat surprising.) Recall that, for file read/write micro-benchmarks, vTurbo does *not* improve disk read throughput much. Yet we observe significant improvement in NFS read and re-read throughput. After some investigation, we figure out the reasons for the improvements here: First, NFS utilizes *pre-fetching* for sequential read operations where multiple read operations are issued in advance. Second, Linux NFS implementation uses *in-kernel* data transfer from files to sockets. As such, the server process is able to process many read requests while the regular vCPU is scheduled and to delegate

the actual file block transfer operations to the kernel threads run by the turbo vCPU, hence achieving much higher throughput.

Secure copy (SCP) throughput SCP involves both CPU activity (for encryption and decryption of data) and I/O activity. We copy a 1GB file using SCP from a client to a VM sharing a core with 2, 3, or 4 other VMs. In this experiment the *sshd* process which is receiving the file is scheduled at the regular vCPU while *both* TCP processing and disk I/O handling threads are scheduled at the turbo vCPU. Figure 3.9(b) shows that vTurbo improves SCP throughput by 53% to 66%.

Table 3.2.: Results from Apache Olio experiment (single- and two-instance)

	Single Instance		Two Simultaneous Instances			
			Instance 1		Instance 2	
Operation	Count Vanilla Xen	Count vTurbo	Count Vanilla Xen	Count vTurbo	Count Vanilla Xen	Count vTurbo
HomePage	4028	5602	3918	5334	3839	5311
Login	1629	2190	1524	2121	1540	2109
TagSearch	5183	7198	4888	6822	4892	6778
EventDetail	3856	5274	3701	5075	3630	5013
PersonDetail	405	562	379	550	381	508
AddPerson	127	178	131	177	120	167
AddEvent	300	402	280	416	279	413
Total	15528	21406	14821	20495	14681	20299
Rate(ops/sec)	51.8	71.3	49.4	68.3	48.9	67.7
Improvement (%)	-	**37.6%**	**-**	**38.2%**	**-**	**38.4%**

Apache Olio To assess the benefit of vTurbo to a cloud application, we use Apache Olio, an event calender developed using Web 2.0 technologies. The Apache Olio benchmark consists of 3 components: (1) a web server to process user requests, (2) a MySQL database

server to store user profiles and event information, and (3) an NFS server to store images and documents specific to events. We use the PHP version of the benchmark.

In our setup, we host the 3 Olio components in 3 different VMs each in a separate physical host. In each host we pin the Olio VM's vCPU to a single core, which is shared by 3 other VMs having 20% of CPU load. We stress the Olio service with 400 client threads generating requests using the Faban client simulator for 6 minutes.

In Table 3.2, the "Single Instance" (2^{nd} and 3^{rd}) columns show the breakdown of total operations (averaged over 3 runs) performed by Olio on vanilla Xen and on vTurbo, respectively. vTurbo achieves higher operation counts than vanilla Xen for all types of operations during the same period. This is because vTurbo improves communication performance among the three Olio components as well as file write performance of MySQL and NFS servers. With vTurbo the overall throughput of the Olio service is improved from 51.8 ops/second to 71.3 ops/second – a 37.6% improvement.

Next, we evaluate the performance of *two* simultaneous instances of Olio, with the same set of components hosted by the same physical servers. In this experiment, of the 4 CPU cores of each server, we dedicate one core to domain0 and one core as the turbo core shared by all VMs. In our replicated Olio configuration, we pin the two copies of each Olio component to the 2 remaining cores respectively, with each core shared by 3 other VMs. Columns 4, 5, 6, 7 of Table 3.2 show the breakdown of total operations performed by the two Olio instances, which are started at the same time and run for the same 6-minute period. Compared with the "Single Instance" results, most rows see a slight reduction of operation throughput for both vanilla Xen and vTurbo configurations. We believe this is due to the sharing of resources such as the disk and network. However, we observe that with vTurbo, the overall Olio throughput is increased by 38.2% and 38.4% for instances 1 and 2, respectively.

4 EFFICIENT DATA ACCESS FOR HADOOP IN VIRTUALIZED CLOUDS

We have demostrated two efficient methods for the CPU sharing scenario. In fact, even we give each VM dedicated CPU cores, the I/O intensive applications still suffer in virtualized clouds because of the device virtualization overhead. For example, many bigdata applications such as Hadoop can not get the same performance after simply being moved to virtualized clouds, because these applications are originally designed for physical machine. To bridge the bigdata applications to virtual clouds, we will present a new data access method for Hadoop in this chapter.

Virtualization introduces a significant amount of overhead to I/O intensive applications due to device virtualization and VMs or I/O threads scheduling delay. In particular, device virtualization causes significant CPU overhead as I/O data needs to be moved across several protection boundaries. We observe that such overhead especially affects the I/O performance of the Hadoop distributed file system (HDFS). In fact, data read from an HDFS datanode VM must go through virtual devices multiple times — incurring non-negligible virtualization overhead — even though both client VM and datanode VM may be running on the same machine. In this paper, we propose vRead, a programmable framework which connects I/O flows from HDFS applications directly to their data. vRead enables direct "reads" to the disk images of datanode VMs from the hypervisor. By doing so, vRead can significantly avoid device virtualization overhead, resulting in improved I/O throughput as well as CPU savings for Hadoop workloads and other applications relying on HDFS.

4.1 Introduction

Many enterprises are increasingly moving their applications from traditional infrastructures to private/public cloud platforms in order to reduce application running costs, both in terms of capital as well as operational expenditure. Cloud providers generate revenue by

keeping their operational costs low while providing good performance for their "tenants". The key technology which drives cloud computing is virtualization. In addition to enabling multi-tenancy in cloud environments, virtualizing hosts in the cloud environment makes resource management increasingly flexible, resulting in significant savings in operational costs.

Similar to other cloud applications, Hadoop [42] applications can also benefit from cloud deployment by taking advantage of the agility to help deploy, run, and manage these clusters while maintaining reasonable performance on par with physical deployments. Compared with running Hadoop on physical machines, virtualized Hadoop allows clusters to be scaled dynamically — separating data and computation in different virtual machines (VMs) while keeping data safe and persistent. Several public and private cloud platforms already embrace this concept. For instance, the Amazon EC2 [1] provides an Elastic Map/Reduce (EMR) service [43] for hosting data processing applications. Similarly, Openstack [44] is developing the Sahara [45] platform with similar goals as EMR. VMware's Hadoop Virtualization Extention (HVE) [11] goes one step further to enhance Hadoop's topology awareness on virtualized platforms (upstreamed into Apache Hadoop release 1.2.0+).

However, running Hadoop inside VMs can lead to sub-optimal performance due to virtualization and data movement overheads. Specifically, the performance of Hadoop inside VMs is heavily dependent on the I/O efficiency of the Hadoop distributed file system (HDFS) [9], because all consumed data by big data applications is first loaded from HDFS. In general, when the client application requests the HDFS datanode to read a file, it reads that file from the local disk and sends its content back to the client over a TCP socket. Depending on the location of the datanode in relation to the client, this performance can vary drastically. For instance, if there is a co-located datanode, standard Hadoop implementations prefer a *local read* from the co-located datanode over other replicas elsewhere. While the local read is efficient when Hadoop is run in non-virtualized environments, its performance can suffer when the client and datanode are co-located on the same physical host but in different VMs (recommended deployments by Docker [10] and VMware's

HVE [11, 12]), due to device virtualization overheads and data movement through protection boundaries (hypervisor, OS, application). *Remote reads* are even slower because of the additional network data transfer overheads.

In particular, the data flow for each file read on HDFS in a virtualized cloud causes data to pass through the virtual devices (e.g., virtual block and virtual NIC) multiple times — causing excessive CPU consumption and performance degradation compared to running Hadoop in physical machines. Further, even if high speed storage hardware (e.g. SSD) is used in the virtualized hosts, the HDFS performance, in terms of throughput and latency, will still be degraded due to a lack of CPU cycles to copy the data. In addition, if low-power processors (Atom, ARM, etc.) are used (as is the trend in some data centers to obtain better per-watt performance), this degradation is even more serious.

Therefore, if we can provide an efficient data movement channel between datanode VMs and client VMs, then we can mitigate the negative impact caused by device virtualization overhead and achieve better data read performance.

In this paper, we focus on improving the I/O performance of virtualized Hadoop applications or other big data applications which rely on HDFS that involve significant data reads, either partially or completely, in their work flows. More specifically, we target data movement between datanode VMs and client VMs — without performing transformation over virtual network and virtual disk. We propose to alleviate the involved device virtualization overheads by enabling HDFS client VMs to directly read data from the co-located datanode VM's virtual disk or utilizing RDMA [46] over converged Ethernet (RoCE) [47] to transfer data directly from the remote disk to the memory of client VMs via its zero-copy networking behavior. By doing so, we are able to reduce 1) device virtualization overhead such as copying data through the virtual disk, virtual network, and the network stack in both the datanode VM and client VM, 2) data copy overhead between the guest kernel/application memory and the data buffers in the host kernel for *remote reads*, and 3) I/O threads scheduling and synchronization overheads caused by "indirect" reads unnecessarily involving the virtual network between VMs.

To realize the idea of an efficient data movement channel in the hypervisor layer, we have developed a system called vRead, where data needed by the HDFS client VM is directly read from the virtual disk of a datanode VM — avoiding unnecessary data copies involved in virtual I/O behaviors. vRead installs a kernel module and a library in the guest providing the file operations interface and a daemon in the host to read data owned by datanode VMs from local and remote physical disks (via RDMA) then map it into the guest memory for the application's use. vRead is transparent to user level applications (such as Hadoop MapReduce, Hbase, and Hive) using HDFS. Therefore, it is able to support all existing applications storing data in HDFS.

To summarize, our contributions in this paper are:

1. We propose a new file operation interface for HDFS client VMs which allows Hadoop applications to read data from HDFS more efficiently.
2. We develop the vRead system, which provides I/O shortcuts at the hypervisor level via components in the guest and in the hypervisor. vRead works for both *virtual local read* (read from co-located datanode VMs) and *remote read*.
3. We present evaluation results from a vRead prototype implemented on KVM. Our microbenchmark results show that vRead achieves higher read throughput, lower latency, and less CPU cycle consumption compared to standard HDFS running on VMs. For example, Hadoop's throughput can be improved by up to 60% for read and 150% for re-read. Results from a number of Hadoop benchmarks also show significant application-level performance improvements with vRead.

4.2 Motivation

In this section, we motivate the problem by demonstrating the impact of virtualization overheads on Hadoop I/O efficiency. We then discuss the inadequacy of existing solutions.

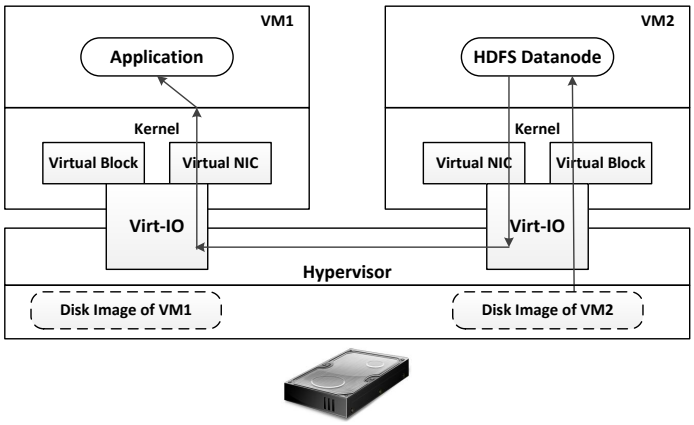


Figure 4.1.: I/O flow in Hadoop for co-located VMs.

4.2.1 Problem Analysis

Virtualization-based overheads (e.g., device virtualization and VM or I/O thread scheduling) cause serious performance degradation to HDFS, and this prevents Hadoop and other applications which rely on HDFS (e.g. Hbase, Hive, and generic Java applications storing data in HDFS) from achieving their expected performance. To illustrate this problem, Figure 4.1 presents a concrete example of a Hadoop application hosted in a VM reading a file from a co-located VM hosting the HDFS datanode[1]. In this scenario, the Hadoop application first creates a TCP socket, connects to the HDFS datanode, and sends the file read request via this socket. Then the HDFS datanode reads the requested file from disk and sends its content back over the same TCP connection.

Even though the virtual disk and virtual network between co-located VMs are very fast (mainly due to inter-VM para-virtual I/O techniques such as virt-io [48] and vhost), this single I/O flow involves *at least 5 data copies*: 3 data copies caused by virt-io, 1 inter-VM data copy, and 1 copy between the kernel buffer and application buffer in VM1 (which may also happen in VM2). Note that each data copy consumes non-negligible CPU cycles and the whole data transfer incurs overhead from both VMs' network stacks. Further, if the file being read was located on a remote datanode VM running on another physical machine,

[1]Such virtual local reads from co-located VMs are more common than remote reads due to existing virtual Hadoop optimizations.

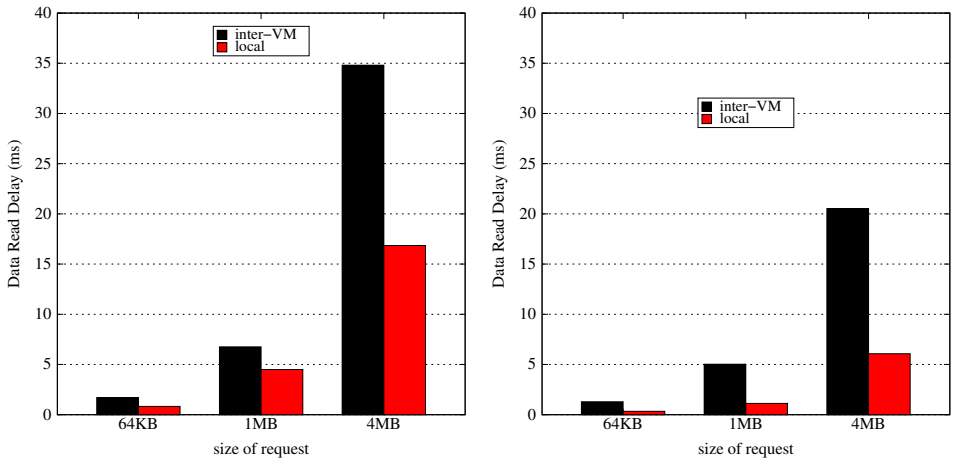
then we would need to also consider the physical networking overhead and additional delays in the host kernels' network stacks on each physical machine. Intuitively, such high I/O costs mean *less CPU cycles for the real Hadoop workload*, which negatively impacts the performance of Hadoop applications.

To illustrate this performance degradation, Figure 4.2 compares the observed read delays from HDFS versus the local file system in a virtualized host. In this experiment, we ran a Java application in one VM that reads a file from the local file system and an HDFS co-located datanode VM. The local file read (i.e., the baseline reading performance) only involves 2 data copies: 1) from disk to guest kernel buffer and 2) from guest kernel buffer to guest user space. We varied the request size (application buffer size) from 64KB to 4MB and used two different read patterns. "Read without cache" means reading data after clearing the disk memory buffer in the guest kernel (virtual disk cache in the hypervisor is disabled). "Read with cache" (or re-read) means reading data without clearing the cache. Figure 4.2 shows that the delay of HDFS hosted in a co-located VM is significantly higher than the baseline read for all cases. The root cause of this result is that inter-VM reads involve *more data copies* and suffer *more device virtualization overhead*.

However, besides the additional data copies, there exists a second, more systemic cause of this performance degradation: *I/O thread synchronization* in the virtualized host. Most existing hypervisors perform I/O (network or disk) in a dedicated per-VM thread that is optimized for I/O performance. For instance, Xen uses a netback thread to process I/O requests for the virtual network, and similarly KVM uses a vhost-net thread for the same task. Therefore, to get good I/O performance, the VM and the corresponding I/O thread *must run cooperatively on different cores* so that the synchronization delay between them is short. If not, context switches between them will cause slow-down at the hypervisor level.

In addition to the synchronization between a VM and its I/O thread, data movement between 2 co-located VMs requires the I/O threads of *both VMs* to synchronize as well. Therefore, we would need 4 free cores to allow 2 I/O VMs to communicate with each other unimpededly. As more VMs run in the same host, the data transfer between VMs is further degraded because the VM scheduler cannot find enough free cores to run the cooperating

threads. Figure 4.3 highlights this problem. In this experiment, we ran 2 co-located VMs hosting a netperf [41] server and client respectively in a quad-core machine. When there are no other active VMs running, we can get high transaction rates, even with varying request sizes. However, if an additional 2 VMs are running CPU-intensive workloads (85% lookbusy [25]) in the same host, then the TCP transaction rate drops by 20%. Since the total CPU utilization of the vCPU thread and I/O thread of each VM hosting netperf is *less than 75%* , we know the host is not overloaded for the 4 VMs scenario. Thus, the only reason for the drop in transaction rate is the synchronization delay of VMs and I/O threads. Again, because virtualized Hadoop requires many such cross-VM data movements, this same scenario causes a loss in I/O performance in virtualized Hadoop as well.



(a) Access delay without cache.

(b) Access delay with cache.

Figure 4.2.: Virtual HDFS data access delay caused by device virtualization overhead.

4.2.2 Alternative Solutions

We now examine several alternative solutions and their shortcomings when used with virtual Hadoop.

HDFS Short-Circuit Local Reads *HDFS Short-Circuit Local Reads* (HDFS-2246 and HDFS-347) [49] allow a read to bypass the datanode process — so that the client to read

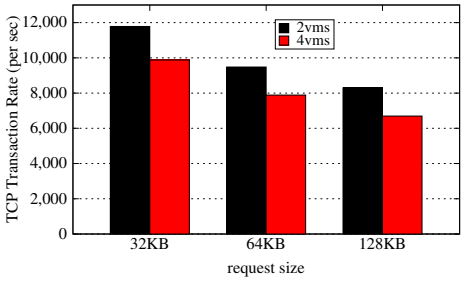


Figure 4.3.: I/O Threads synchronization overhead.

each file directly. This approach is only possible when the client process and the datanode process execute in the same *operating system* (OS). However, virtual Hadoop separates HDFS clients and datanodes in different VMs to obtain a more scalable Hadoop cluster and better on-demand resource allocation. Further, locating some clients and datanodes in the same VM to utilize *HDFS Short-Circuit Local Reads* would cause a significant penalty to *virtual remote reads* (inter-VM data reads). Thus, traditional *HDFS Short-Circuit Local Reads* does not work for virtual Hadoop.

VirtFS *VirtFS* [50] is a para-virtualized file system interface designed to improve pass-through technologies which rely on the virt-io framework and the 9P protocol. With VirtFS, a guest can easily exchange data with the host. However, there are several aspects of VirtFS which impedes its useful application to HDFS. First, the 9P protocol used by VirtFS is not efficient, resulting in unacceptable disk I/O performance. Second, the explicit shared directory assignment of VirtFS makes the virtual Hadoop cluster setup more complex and inflexible. Third, it is not applicable when datanodes and clients are in different physical machines, which is a typical pattern in distributed Hadoop systems.

Hadoop Virtualization Extensions VMware's HVE enables virtual Hadoop to know the location of data files in order to co-locate inter-VM data reads. However, it does not optimize the data read path in the hypervisor (causing excessive data copies). This data flow is similar to the scenario shown in Figure 4.1, and is thus susceptible to the same aforementioned issues.

Inter-VM Shared Memory Inter-VM Shared Memory [51–53] is a popular technology used to boost the performance of inter-VM communication. However, this zero-copy communication between VMs can only reduce one data copy in the data flow of virtual Hadoop (Figure 4.1). The involvement of datanode VMs still imposes additional overheads (i.e., device virtualization and I/O threads synchronization) on each data read performed by applications running in client VMs. Again, this approach only works for co-located VMs running in the same machine.

4.3 Design

Table 4.1.: vRead API.

API Function	Input Parameters	Return Value	Description
vRead_open()	blk_name, datanodeID	vRead descriptor	Open the file for an HDFS block stored in a specified datanode and get the corresponding vRead descriptor.
vRead_read()	vRead descriptor, buffer offset, length	Number of bytes read into buffer	Attempt to read up to *length* bytes from the file pointed to by vRead descriptor.
vRead_seek()	vRead descriptor, offset	Resulting offset as measured in bytes	Set the file offset for an opened file pointed to by the given vRead descriptor.
vRead_close()	vRead descriptor	Successful (0) or not (-1)	Close the file for a given HDFS block indicated by the vRead descriptor.

Based on the above discussion, it is evident that if we provide an efficient file read mechanism for HDFS (i.e. if the HDFS clients can read the disk blocks owned by datanode VMs directly regardless of their location), then we can improve virtual Hadoop's performance significantly. vRead achieves this by letting the HDFS client VMs independently perform data reads directly from the disk instead of channeling it through the datanode VM.

First, vRead enables a shortcut in the I/O path that reduces the data access delay. Second, a reduction of data copies translates to the reduction of CPU consumption, thus more CPU resources are available for the actual Hadoop CPU-bound tasks. Third, a shortened

inter-VM data transfer path significantly reduces the I/O processing delay caused by synchronization of the VMs' I/O threads.

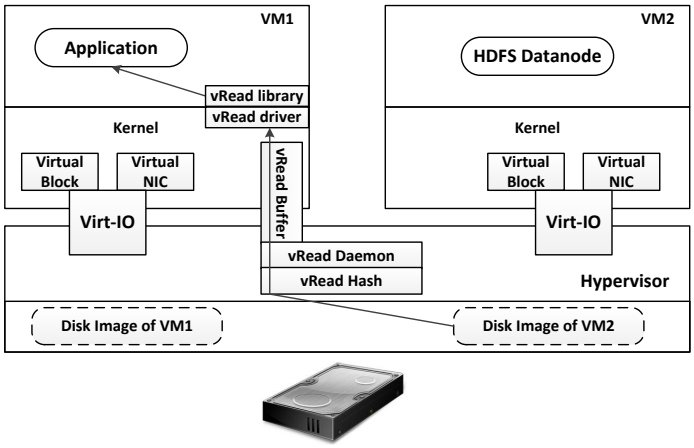


Figure 4.4.: I/O flow in Hadoop for co-located VMs with vRead.

To enable an HDFS client VM to read data directly from a datanode VM's virtual drive, vRead needs to provide three features: 1) a new user-level block read interface for inter-VM block reads, 2) a mechanism to directly read from the virtual drive of the datanode (an active VM) without channeling through the datanode and, 3) an efficient data sharing (zero-copy) and communication channel between the guest OS and hypervisor.

The architecture of vRead is shown by Figure 4.4 and Figure 4.5. The above requirements are realized with the three main components of vRead shown in these figures. In the following subsections we will discuss each of these sub-components of vRead in detail.

In this paper, vRead targets HDFS but this framework is able to be generalized to other similar distributed file systems such as QFS [54] and GFS [55].

4.3.1 vRead User-level API

vRead provides a set of user-level library functions for HDFS to use. It should be noted that these API calls will only be used by the HDFS components. Hadoop applications are unaware of the existence of the vRead system and continue to use the original HDFS interfaces to read data — hence requiring no modifications to them. vRead's implementation

is independent of the guest OS so that this user-level library does not need to be adjusted or changed whenever a different OS is used. We outline the vRead user-level APIs in Table 4.1. The vRead APIs are a set of functions provided in a user-level library (*libvread*) which hides the complexity of interacting with the underlying vRead components. This library provides 4 main functions. To read a file stored in a datanode VM's virtual disk, we first need to call *vRead_open* to initialize a set of data structures inside both the guest OS as well as inside the hypervisor. This will return a vRead descriptor which is used as a parameter in the rest of the functions. HDFS only understands block names, hence the vRead descriptor is invisible to it. Thus, each obtained descriptor is stored in a hash table in the user-level library, which maps the block names to vRead descriptors, until the *vRead_close* function is called. This lets HDFS reuse the descriptor for subsequent read/seek operations on the same block file.

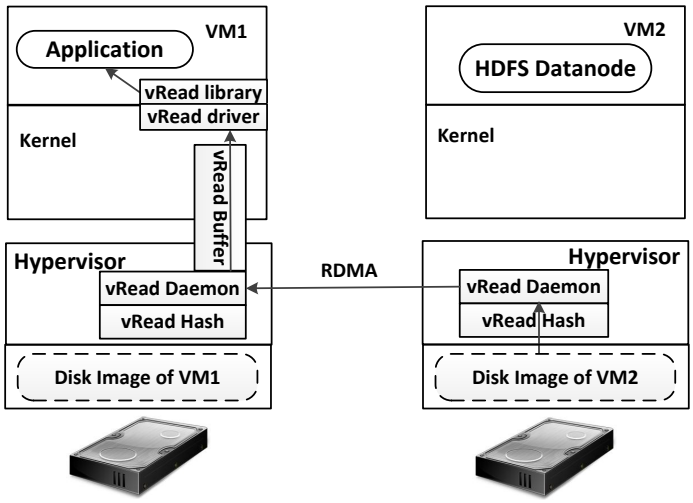


Figure 4.5.: I/O flow in Hadoop for remote VMs with vRead.

HDFS is designed to support very large files in a distributed environment. Thus, each file stored in HDFS is divided into chunks called data blocks (some smaller files on disk, 64MB each by default), and if possible, all blocks will reside on different datanodes. As a distributed file system, all actual data (HDFS blocks) are stored in the same path in each datanode. To read a file from HDFS, the Hadoop client needs the assistance of the namen-

ode which stores the metadata of each file (block mappings, destination data node ID, etc.). In this work, we preserve all logic between clients and the namenode, and only modify the file read logic in the HDFS client interface. We re-implement the HDFS interface with the vRead file operation interface. When an HDFS client plans to read a specific file, it first gets the block list from a namenode then uses the vRead interface wrapped by the HDFS interface to send target block information (block name and datanode ID) and operations to the vRead per-VM daemon running in the hypervisor. The daemon then reads the data from the destination virtual disk and returns to the client. We discuss the operation of this daemon in the following subsection.

4.3.2 Reading from a Datanode's VM Disk Image

We use a daemon running in the hypervisor to aid in reading from the virtual disk images of datanodes. This daemon receives the requests from a guest and uses a hash table to store the mappings between HDFS datanode IDs and the corresponding virtual disk (which can be a local image file, NFS, or iSCSI) information for each datanode VM. This hash table is initialized when Hadoop is started by accepting information from the namenode and looking up the VM's configuration. For the datanode VMs running on other machines, we only store the IP address of the destination host machine. This table is dynamically updated once there are any datanode VMs created, deleted, or migrated.

Reading from a Local Datanode To read data from the datanode VM's virtual disk (i.e., an image file) in the hypervisor using existing POSIX APIs, vRead has to meet the following two requirements. First, the vRead hypervisor daemon should be able to understand the file system in the virtual disk. This is because the HDFS blocks are stored as regular files in the datanode's file system and accessing these files requires the hypervisor layer to understand the file system layout of the datanode VM. In KVM environments, a Linux kernel functions as the hypervisor and this kernel can interpret most file systems used by guest OSs (typically guests also use a similar if not the same Linux version).

To access the content of the datanode's file system, all datanode VMs' virtual disks are mounted in read only mode to a specific directory in the hypervisor (e.g. /mnt/datanode1) as loop devices with the assistance of *losetup* (qemu-nbd module is needed if the disk image file is qcow format). Since each virtual disk is installed independently of the file systems (maintained by the guest OS), we need to read the partition tables on these virtual devices and create device maps over the partitions' segments (we use the existing Linux tool *kpartx* for this).

Second, we need to synchronize accesses to this file system by the datanode and HDFS clients. The datanode's access is read-write while the HDFS clients' accesses are read-only. Since the file system within the guest OS is independent of the file system in the hypervisor (file system of the VM is in the VM's address space and hence opaque to the hypervisor unless we use VM introspection tools), new HDFS blocks generated by the datanode are invisible to the vRead daemon. Thus, we need to refresh the directory entry and inode cache information of the hypervisor mount point of the datanode's disk partition if any HDFS blocks are created, deleted, or renamed. However, HDFS mostly operates in "append-only" mode, hence we do not need to refresh all information for the mount point. Only the added inodes (i.e., new files representing the HDFS blocks) need to be updated.

The synchronization is achieved through the Hadoop namenode. When a datanode writes new blocks to the file system, it notifies the namenode about the availability of the new blocks so the readers (HDFS clients) can access these blocks. We use this notification as a trigger to refresh the mount point corresponding to that datanode. The new block information is obtained from the *vfsmount* structure and *superblock* of the corresponding virtual disk. The whole process is similar to a remount. This process is sufficient to guarantee that there is no read/write conflict issues since HDFS follows the *write-once and read-many* approach for its data blocks. This approach assumes that a file in HDFS will not be modified once it is written. All new written data will generate new blocks. So we do not need to be concerned with read/write conflicts when issuing a direct read on a virtual disk without notifying the owner VM.

Reading from a Remote Datanode To read data from a remote datanode, the local vRead daemon contacts the remote host's vRead daemon using RDMA and sends the request to the remote daemon. The remote daemon then performs the read operation on the local disk (as discussed above) and returns the data via RDMA. For vRead, RDMA is not necessary (traditional TCP/IP also works), but RDMA helps vRead consume less CPU cycles and ensure lower latency for remote data reads.

RDMA allows an application to communicate directly with another application via remote memory read/write. This means that an application does not need to rely on the operating system to transfer messages. To communicate with the remote end, we (1) Register a Memory Region (MR). This Memory Region is similar to the packets buffer, but it is shared by the remote party which can directly access it via the RDMA device. (2) Create a Send and a Receive Completion Queue (CQ). (3) Create a Queue Pair (QP) that is a Send/Receive Queue Pair. To send or receive messages, Work Requests (WRs) are placed onto a QP. When processing is completed, a Work Completion (WC) entry is optionally placed onto a CQ associated with the work queue. To enable the RDMA, we use the support of two libraries (rdmacm and libverbs) in userspace and a physical NIC with RDMA support. Traditional RDMA requires an infiniband network which is very expensive, so we use RoCE (RDMA over Converged Ethernet) instead of infiniband.

4.3.3 Data Sharing and Communication Channel

The communication between the guest OS and local vRead daemon is accomplished using a shared memory channel and an event mechanism. This channel is a memory ring buffer shared by the VM and hypervisor. For each HDFS read request, the vRead driver in the guest places a request in the shared memory and fires an event to notify the vRead daemon in the hypervisor. In turn, the vRead daemon in the hypervisor reads the data from the datanode VM's disk image, writes the data to this channel, and then sends an event to the guest OS so the data can be consumed by client applications. We cannot use KVM virt-io's ring buffer for this purpose since the HDFS client VM needs to read data from

the datanode VM's virtual disk and such an I/O operation is not the intended use of virt-io channels. Instead we have to use a shared memory mechanism between the local vRead daemon and guest OS's vRead driver.

To achieve zero-copy between the hypervisor and guest OS in the VM, we use a POSIX SHM object as the shared buffer that is assigned to each VM as a character device. For each guest, this shared memory object appears as a virtual PCI device inside the guest OS. This POSIX SHM object is divided into multiple chunks (default 1024) to comprise a ring buffer. The vRead daemon uses SysV APIs to read from/write to this shared buffer. The guest maps the virtual PCI device's address space to it's own address space and then performs read/write operations. The synchronization between the vRead daemon and guest OS is guaranteed by a read/write lock on each chunk.

To send notifications between the vRead daemon in the hypervisor and guest OS, vRead uses interrupts between them that are implemented by assigned *eventfds*. Each VM listens on its own eventfd, and uses its corresponding vRead daemon's eventfd to send an event (and vice versa). The only difference is that the vRead daemon operates the event directly, whereas the event received by the VM has to be translated into a virtual interrupt which can be recognized by the guest OS. This translation is done by the vRead driver in the guest kernel. With this channel, applications in the guest OS can send requests (e.g., read a HDFS block) to the vRead daemon and get the result from the shared memory buffer.

4.4 Implementation

We have implemented a prototype of vRead for the KVM [56] hypervisor. We used Linux 3.12 as the kernel of the VMs and the KVM host. The Hadoop version is 1.2.1.

vRead includes new implementations of the read interfaces for an HDFS client (in the *DFSClient* class). These interfaces mainly contain *read*, *seek*, and *skip* functions located in the *DFSInputStream* class (a subclass of *DFSClient*). Of these functions, the *read* function is the most important as it is frequently called during HDFS reads. The *DFSInputStream* class has 2 different *read* functions that vRead overrides: called *read1* and *read2* in this

paper. *read1* reads a large file from the beginning and its request size is smaller than one HDFS block (e.g., for use by applications performing sequential reads). Its vRead implementation is shown in Algorithm 4. Before reading an HDFS block, vRead checks whether the corresponding file has been opened previously (and thus has a corresponding *vfd* in hash) or not. If the file has not been opened previously, a new vRead descriptor *vfd* is created by calling *vRead_open()* and added to a hash table for future use. Upon subsequent calls to *read1*, vRead checks if the input descriptor is a valid vRead descriptor (i.e., in the hash table). If so, it is used to read data via *vRead_read()*; if not, the original HDFS function *read_buffer* is called to perform the read from the datanode. *read2* reads data from a specific position in a file (e.g., for use in asynchronous/random reads). The implementation of *read2* is outlined by Algorithm 5. Generally, *read2* is similar to *read1* except that it is allowed to read across multiple blocks so vRead has to collect all involved block information from the namenode and perform the *vRead_read* on them one by one.

Additionally, we slightly modify the write interfaces of HDFS to update the dentry/inode of the mount point for new blocks generated by the datanode. Specifically, we call the *vRead_update* function at the end of the standard *append* function (in the *DFSOutputStream* class) once a full block is written to the datanode VM. Likewise, the same thing happens for a block delete or rename. Note that we do not have to call *vRead_update* for each *append* operation before a new block is completely created. Since all vRead functions in *libvread* are written in C, but HDFS is implemented in Java, all vRead functions have to be called via a Java native interface (JNI). After adding the vRead extensions to the *DFSClient* class, the Hadoop source code was re-compiled and we replaced the hadoop-core-1.2.1.jar required by the Hadoop running environment with our new one.

To interact with the vRead buffer, we implemented a guest kernel driver that: 1) helps the guest OS recognize the assigned POSIX SHM object as a virtual PCI device and 2) translates the eventfd signals to virtual interrupts and vice versa. This driver is a loadable kernel module whose implementation is based on the ivshmem [52] VM driver. The address of the virtual PCI device representing the vRead buffer is mapped to the address space via *mmap()* — so that applications in the guest can read from/write to this ring buffer by calling

the vRead series functions in *libvread*. The vRead ring buffer is divided into 1024 slots (the size is configurable, with a default of 4KB) comprising the critical area between the application thread in the guest OS and the vRead daemon in the hypervisor. A spinlock (*pthread_spinlock_t*) is used on each slot to guarantee synchronization safety.

The vRead daemon is a generic thread granted read privilege to the entire local physical disk of the hypervisor. In the KVM platform, each VM is a process/thread in the host. Therefore, the vRead daemon can communicate with the process representing a VM via an eventfd and a read/write on the shared POSIX SHM object (vRead buffer).

To connect to remote vRead daemons on other machines with low latency and low CPU cost, we use RDMA interfaces (declared in rdma/rdma_cma.h and infiniband/arch.h) instead of TCP/IP APIs to exchange data[2]. Specifically, we call a few standard infiniband verbs such as *ibv_reg_mr* (register memory regions), *ibv_post_send* and *ibv_post_recv* (send and receive requests) on the Ethernet links via RoCE techniques to directly map the working set address of request/response to the remote memory.

To update the file system for new blocks added in a mounted virtual disk, vRead needs to refresh the dentry/inode of the mount point if the *vRead_update* function is called in the guest OS. This is done by calling a function extended from *attach_recursive_mnt()* (in the source code of the mount command) which is responsible for updating the *vfsmount* structure of the host file system.

4.5 Evaluation

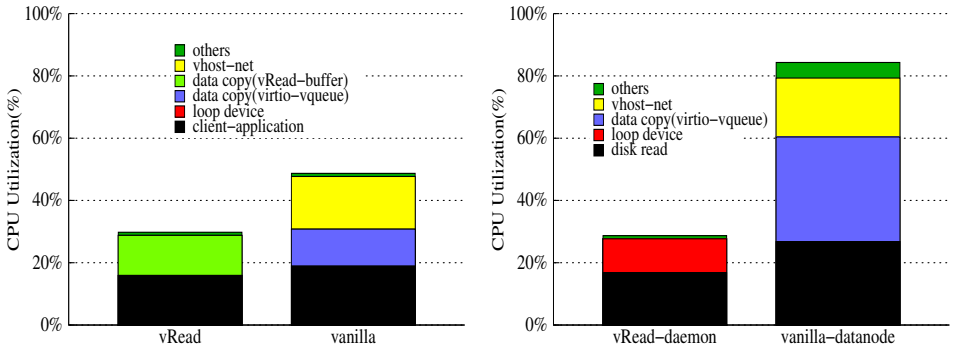
This section presents our evaluation of vRead using both microbenchmarks and real world Hadoop applications.

Evaluation Setup Our testbed consists of multiple servers, each with a 3.2 GHz Intel Xeon quad-core CPU and 16GB of memory. An SSD and 10Gbps RoCE NIC are installed in each server. All physical servers are connected by 10Gbps network in a LAN. These servers run KVM as the hypervisor and Linux 3.12 as the OS for all guest VMs and the

[2]We also implemented a TCP/IP version prototype, but note that it consumes more CPU cycles for remote reads

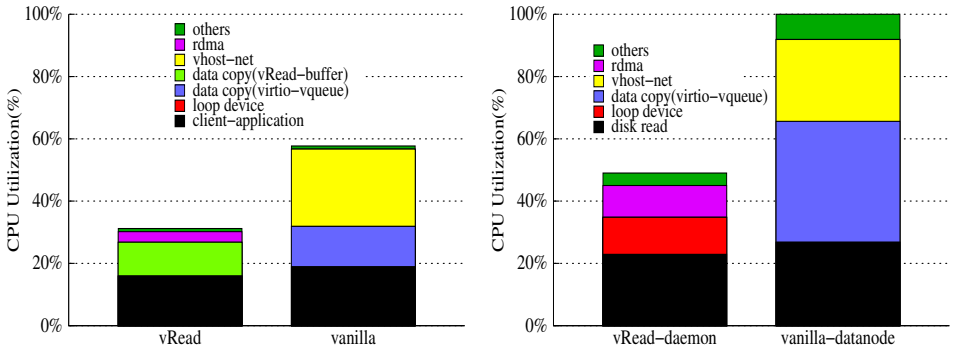
hosts. The Hadoop version is 1.2.1. To emulate the different CPUs (low power and high frequency), the frequency of our Xeon processor is set to different values (1.6 GHZ, 2.0 GHZ and 3.2 GHZ) via the *cpufreq-set* command [57].

All VMs in our experiments are assigned 1 vCPU and 2GB RAM each. We do not set the CPU affinity for the VMs. KVM vhost-net is enabled to boost the virtual network performance. vhost-blk is disabled because it is still the test version in the latest KVM release. The virtual disk image of each VM is a raw image file located in the local SSD.



(a) Client CPU utilization. (b) Datanode CPU utilization.

Figure 4.6.: CPU utilization for co-located read.



(a) Client CPU utilization. (b) Datanode CPU utilization.

Figure 4.7.: CPU utilization for remote read with RDMA.

4.5.1 Microbenchmark Performance

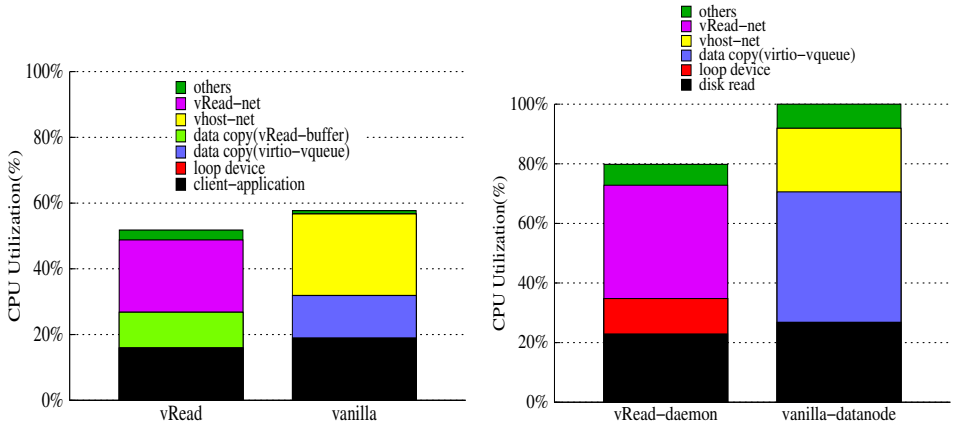
CPU Savings To verify whether vRead's shortcut to file reading can reduce overall CPU cost or not, we compare the CPU utilization of reading a 1GB file from the HDFS with vRead and without. The request size (application buffer on the client-side) of each read is 1MB. There are three scenarios: 1) the client VM and HDFS datanode VM are running on the same machine (i.e., co-located scenario), 2) the client VM and datanode VM are running on two different machines (i.e., remote scenario) and the vRead daemons use RDMA to exchange data, and 3) still the remote scenario but the vRead daemons use TCP instead of RDMA to exchange data.

Figure 4.6 shows the average CPU utilization when the client reads from the co-located datanode VM. As expected, the VMs' CPU utilization with vRead is much lower than the vanilla case. Since there is no virtual network involved in vRead for this case, vRead saves a significant number of CPU cycles both in the guest and host. The direct data read from disk also avoids any unnecessary data copies between the 2 VMS, between the host and datanode VM, and between the guest kernel buffer and application buffer in the datanode VM. In total, we save around 40% of the CPU cycles on the client side and around 65% on the datanode side with vRead.

The results of the remote-read scenario with RDMA enabled are presented in Figure 4.7. vRead still beats the vanilla case on both the client and datanode sides. Thanks to RDMA, the inter-host network cost of vRead (shown by the *rdma* bar) is far lower than the vanilla (shown by the *vhost-net* bar). Since our prototype uses an *active model* for RDMA data exchange on the datanode side (actively pushing data into the client's memory), the RDMA cost of the host running the datanode VM is higher than that of the host holding the client VM. In this case, we save around 45% of the CPU cycles on client side and more than 50% on datanode side.

We also evaluate the TCP version of the data exchange for remote reads. In this setup, the vRead daemons running on different machines use TCP/IP interfaces instead of RDMA verbs to exchange data. The results of these tests are shown in Figure 4.8. Compared with

the RDMA version, the number of CPU cycles spent in sending/receiving data with TCP is significantly higher. Note that the total CPU utilization is still slightly lower than the vanilla case, which also uses TCP/IP, because it avoids copying data from the host to the datanode VM. Nonetheless, the network processing of the vanilla setup (*vhost-net*) is even more efficient than our TCP component ("vRead-net"). This is because all operations of vhost-net are completely done in kernel space, while our TCP version of vRead is a user-level thread in the host — which has to switch between kernel space and user space and thus consumes more CPU cycles. Therefore, we prefer the RDMA version utilizing the RoCE because it helps achieve encouraging performance with low cost.



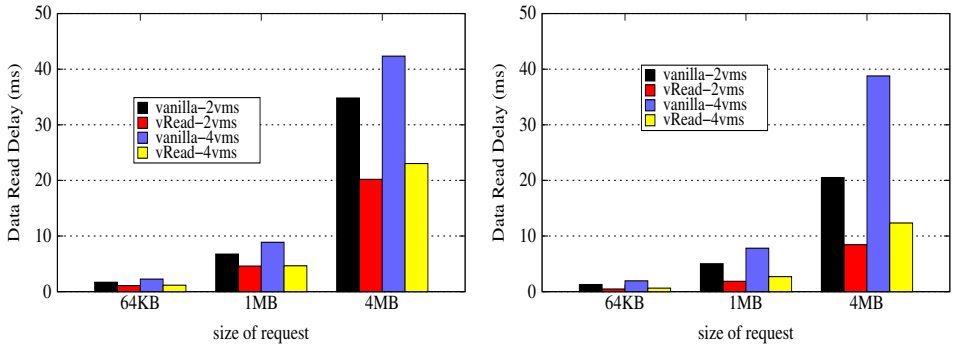
(a) Client CPU utilization with TCP. (b) Datanode CPU utilization with TCP.

Figure 4.8.: CPU utilization fore remote read with TCP.

Data Read Delay Reduction vRead allows the client VM to read files (HDFS blocks) from a datanode VM's disk image directly. Theoretically it would achieve performance close to that of reading data from the local file system. So we repeat the data access delay experiment (shown in Figure 4.2) described in Section 4.2. However, now we replace the *local reads* by HDFS reads with vRead; the baseline is still vanilla HDFS reads. Figure 4.9 shows the average data read delay when performing a 1GB file read from a co-located HDFS datanode VM. The request size varies from 64KB to 4MB. In the first scenario, only 2 VMs (client and datanode VMs) are running in a quad-core machine. The CPU frequency

is set to 2.0 GHZ. We issue two kinds of reads for this case. Figure 4.9(a) shows the results after clearing the memory cache in the datanode VM and host. Figure 4.9(b) shows the results without clearing the memory cache, that is, all data are read from the memory cache and not the disk (called re-read). Our results show that for any request size, vRead beats the vanilla case in both read and re-read evaluations, because it cuts 3 data copies for each read.

Further, recall that I/O thread synchronization may negatively impact inter-VM communication — resulting in HDFS read degradation when CPU competition happens among VMs and I/O threads. To measure vRead's effectiveness in this scenario, we ran an additional 2 VMs in the same quad-core machine so that all vCPU threads and I/O threads *cannot* always find a free core to run on. Hence, the HDFS data read delay in the 4 VMs scenario is higher than the 2 VMs case. vRead's performance is also affected, but its degradation is lower than the vanilla case. Therefore, the gap between vRead and the vanilla case is larger in the 4 VMs scenario. Overall, vRead can reduce the data access delay of the co-located HDFS reads by up to 40% for the 2 VMs scenario and up to 50% for the 4 VMs scenario compared with the vanilla environment.



(a) Data access delay without cache.

(b) Data access delay with cache.

Figure 4.9.: Data access delay for virtual HDFS.

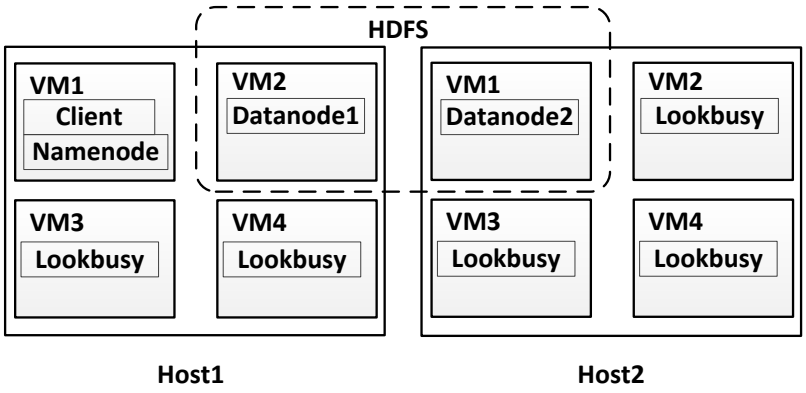
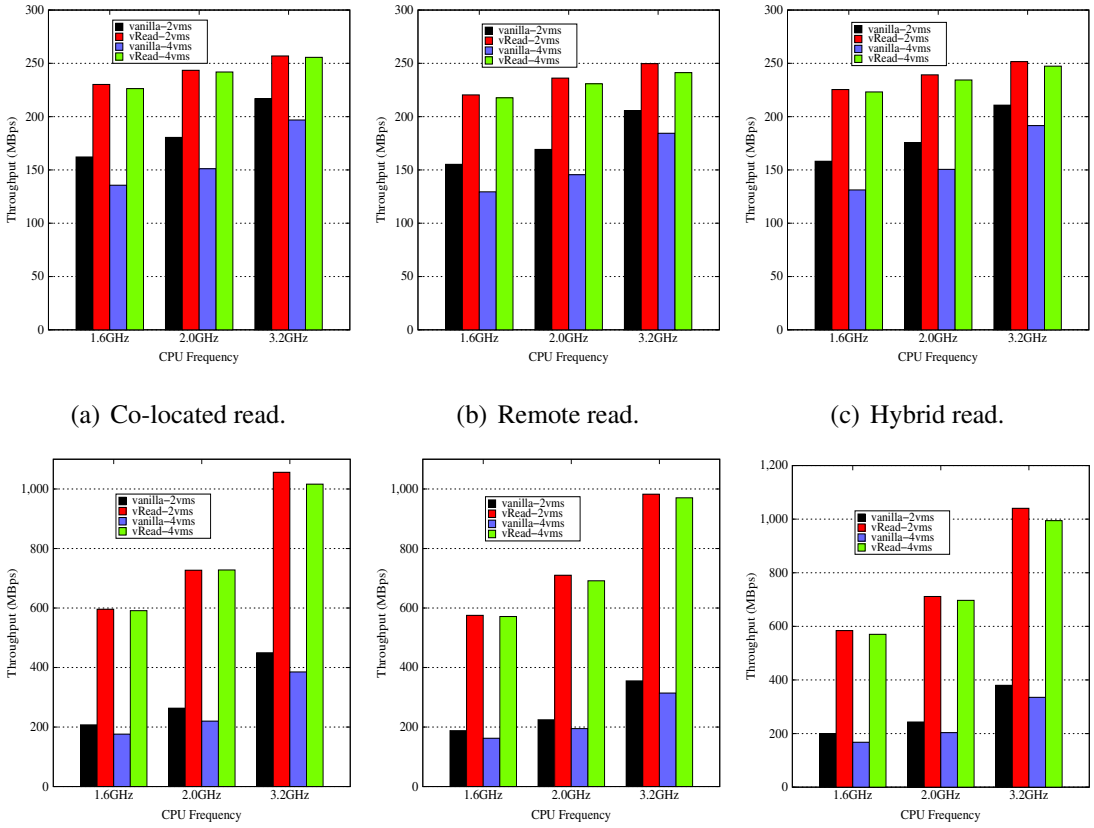


Figure 4.10.: Hadoop setup.

4.5.2 Application Performance

Hadoop Performance In this experiment, we set up a simple Hadoop cluster containing one client VM and two datanode VMs. The namenode resides in the same VM as the client. More specifically, one datanode VM shares the same host (Host1) with the client VM, the other datanode is hosted by another physical machine (Host2) in the same LAN. Each physical machine hosts up to 4 VMs, and the rest of the VMs are background VMs running an 85% lookbusy [25] workload. The setup is shown in Figure 4.10.

In the *virtual local read* scenario, the client reads data from only the co-located datanode VM. In the *remote* scenario, only the data stored in the datanode VM located on Host2 is read. *hybrid* means that the client read data from both the co-located datanode VM and remote datanode VM, which is a more generic scenario in the real world. A widely used HDFS benchmark TestDFSIO is chosen to measure the read throughput of HDFS and the CPU running time. Unlike the simple Java application used in our data access delay experiment, TestDFSIO is a real Hadoop workload utilizing the Map/Reduce framework. In our experiment, the client reads/re-reads 5GB of data from the HDFS each time with the default 1MB memory buffer. To measure the performance on different processors, we vary the CPU frequency from 1.6 GHZ to 3.2 GHZ to emulate low-power processors and powerful processors. The results shown in Figure 4.11 indicate that if only the client VM



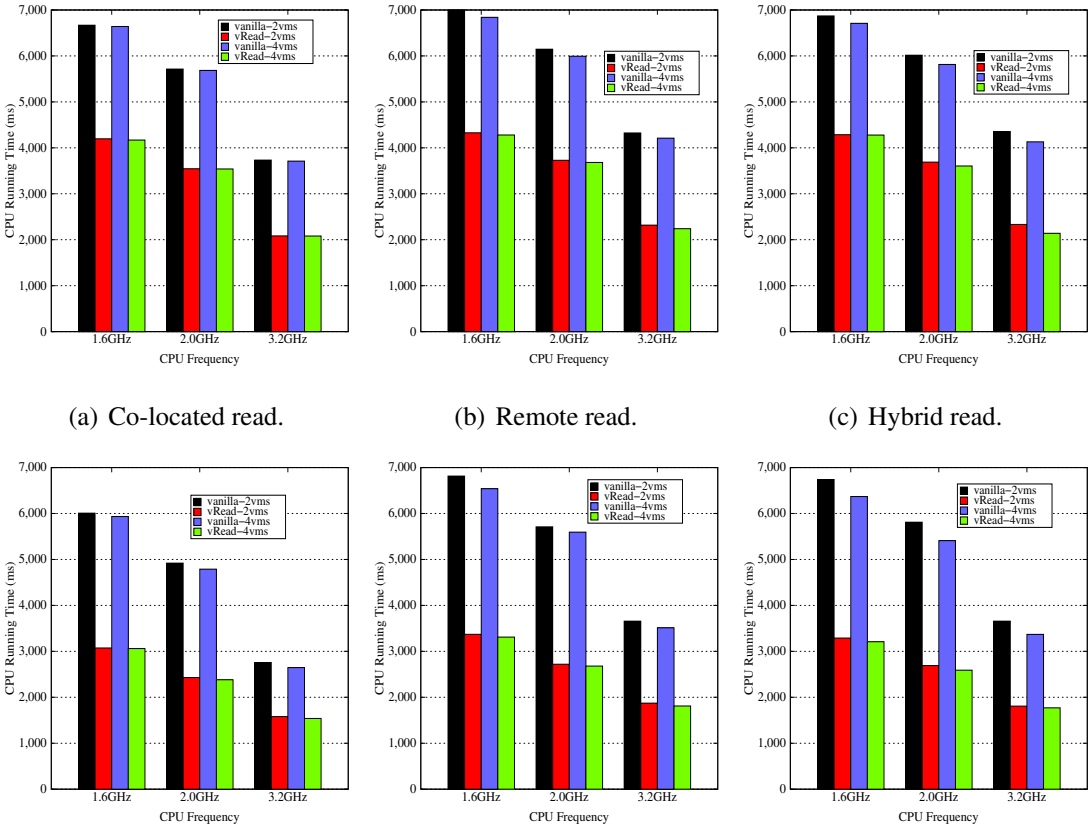
(a) Co-located read. (b) Remote read. (c) Hybrid read.

(d) Co-located re-read. (e) Remote re-read. (f) Hybrid re-read.

Figure 4.11.: HDFS read throughput.

and datanode VM are running (2 VMs scenario) vRead obtains around 20% throughput improvement over the vanilla Hadoop on powerful processors (3.2 GHZ). While, on the low-power processors (1.6 GHZ), the throughput improvement increases to around 41%. The CPU bottleneck on low-power processors becomes more severe for the vanilla case, but its impact on vRead is slight because vRead requires far fewer CPU cycles to perform a read from the HDFS which is verified by our CPU saving experiments.

When 2 or 3 additional VMs hosting 85% *lookbusy* are running on the same hosts (4 VMs in total), each VM and its I/O thread cannot be assured a free core to run on. Thus the synchronization among VMs and their I/O threads are delayed by the CPU fair-share scheduler. This is why the vanilla case's throughput drops by up to 22% for the 4 VMs



(a) Co-located read. (b) Remote read. (c) Hybrid read.

(d) Co-located re-read. (e) Remote re-read. (f) Hybrid re-read.

Figure 4.12.: HDFS read CPU time.

scenario. Whereas, vRead's performance just drops slightly due to less work being done by the I/O threads (i.e., no inter-VM communication). Therefore, vRead has up to 65% improvement over the vanilla case in the 4 VMs scenario. Figure 4.12 shows the actual CPU running time (not the task completion time) spent by the TestDFSIO benchmark when performing the 5GB reads from the HDFS. This shows that vRead still saves significant CPU cycles along with gaining better throughput, which is helpful to reduce the electric power cost for data centers while obtaining encouraging performance.

To enable the new HDFS blocks written into the datanode VM to be visible to the vRead daemon in the hypervisor, we need to update the mount point information once a new file is generated in the file system of the virtual disk image. We verify that this will not hurt the

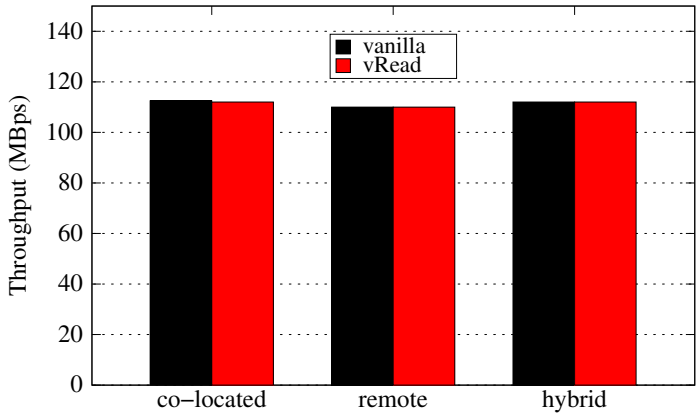


Figure 4.13.: HDFS write throughput.

HDFS write performance by running TestDFSIO-write with the same setup as TestDFSIO-read. Figure 4.13 shows the results of 3 different scenarios (CPU is set to 2.0 GHz) for the vanilla case and vRead. From this figure we can see that the overhead of updating the information of the mount directory is negligible.

Big Data Analysis Tools with vRead There are a number of powerful data query tools in the Apache Hadoop ecosystem helping people store and analyze big data efficiently and safely. In this subsection, we will evaluate the performance of vRead on some of these tools (Hbase [58], Hive [59], and Sqoop [60]).

Table 4.2.: Performance improvement for Hbase.

	Scan	SequentialRead	RandomRead
Vanilla	6.26MB/s	3.01MB/s	2.48MB/s
vRead	7.97MB/s	3.72MB/s	2.91MB/s
% Improvement	**27.3**	**23.6**	**17.3**

HBase Apache HBase is a Hadoop database: a distributed, scalable, big data store. It is capable of hosting very large tables — millions, or even billions, of rows on top of commodity hardware. Each read/write operation is split into Map/Reduce jobs running on the underlying Hadoop clusters. For this experiment, we installed HBase-0.94 on top

of our Hadoop deployment (same as the *hybrid* 4 VMs setup in last subsection). The CPU frequency is set to 2.0 GHZ and the frequency scaling is disabled. In order to use the extended HDFS with vRead, we replace the Hadoop-core-1.2.1.jar under the *hbase-0.94/lib* directory with our new jar package with vRead. We use the built-in HBase benchmark *PerformanceEvaluation* to measure the performance of scan, sequential read, and random read in HBase. We inserted 5 million records via *PerformanceEvaluation-SequentialWrite* to HBase as testing data. The results are shown in Table 4.2. Compared with the vanilla case, vRead can improve the throughput of the 3 operations by 27.3%, 23.6% and 17.3%, respectively.

Hive Apache Hive is a data warehouse software that facilitates querying and managing large datasets residing in distributed storage. Hive provides a mechanism to project structure onto this data and query the data using a SQL-like language. Similar to the HBase test, we installed Hive-1.1 on top of our Hadoop deployment (same as the *hybrid* 4 VMs setup in TestDFSIO test). Following the evaluation approach from the UC Berkeley AMP Lab, we first created a test table in Hive storing some user information (*id, name, birthday, etc.*) and loaded 30 million rows into this table. Then we ran a sql query (*select * from test where id* $\geq$ *x and id* $\leq$ *y*) to select the rows meeting the query conditions. The query completion time is shown in the second column of Table 4.3. From this we can see that a 21.3% time reduction was achieved by vRead.

Table 4.3.: Performance improvement for Hive and Sqoop.

	Select Sql for Hive	Sqoop Export
Vanilla	17.945s	385.136s
vRead	14.117s	342.508s
% Improvement /(Reduction)	**21.3**	**11.3**

Sqoop Apache Sqoop is a tool designed for efficiently transferring bulk data between Apache Hadoop data store and structured datastores such as relational databases (MySQL, Oracle, MSSQL etc.). Here, we measure the performance of exporting data from Hive

to MySQL. The *export* operation, in fact, is a process of reading data from HDFS and inserting them to a relational database. In this experiment, we move the test table storing 30 million rows used in our Hive test to a MySQL database running in another physical machine on the same LAN through *Sqoop-export*. The job completion time is shown in the third column of Table 4.3. For these results, we can see that vRead can reduce the time by around 11.3%. The reduction is lower than our other test case, because the *export* performance is limited by both the read efficiency of HDFS and the insert (write) efficiency of MySQL which vRead cannot optimize.

4.6 Discussion

Interplay with Modern Hardware SR-IOV [61] devices and IOMMUs such as Intel VT-d [62] enable the hypervisor to directly assign devices to the guests. This allows the guests to directly interact with the physical devices and eliminates virtualization overhead. However, it does not work for inter-VM data movement, which is common with virtual Hadoop. Additionally, with these hardware, delays caused by synchronization between VMs and I/O threads will still impact the I/O performance of the communicating parties. Actually, vRead is compatible with SR-IOV and IOMMUs because vRead does not modify the networking routines for packets on the outgoing host. Therefore, vRead and those modern hardware are complimentary and could mutually benefit from each other.

Compatibility with VM Migration VM live migration [63] is helpful for maintaining overall load balance among physical servers in a data center. Live migration requires storing the VM disk images in a centralized storage. Hypervisors (or the VM host) access the VM images via NFS or iSCSI. vRead still works in this setup. All image files deployed by NFS or iSCSI can be mounted in the hypervisor's file system. The reads/writes on these virtual disk images are the same as that on image files located on a local disk drive. Once a VM is migrated to another host, the vRead hash tables in both hosts just need to be updated.

Direct Read Bypassing the File System in the Host Since the vRead daemon has the privilege to access all local devices, it can directly read a datanode VM's virtual disk and

bypass the file system in the host. This method can avoid mounting the virtual image files in the host and updating the mount point's dentry/inode for any new blocks. However, the main drawback of this method is that it cannot benefit from the file system cache, that is, all reads have to load data from the physical disk drive. Also, this approach needs to manually translate the address of each file several times (guest logical to guest physical, guest physical to host logical, host logical to host physical) for each read. This is much more complex than mounting the virtual disk image to the host's file system — which allows vRead to use Linux POSIX APIs to read/write files.

Algorithm 4 DFSInputStream read1 with vRead interface

```
1: vfd is the vRead descriptor for a given HDFS block
2: vfd_hash is the hashtable storing the mappings of HDFS block and vfd
3: datanode_id indicates the target datanode
4: blk is the instance of an HDFS block to read
5: buf is the application buffer
6: len is the number of bytes to read
7: off is the offset of the data block
8: procedure READ(buf, off, len)
9:     blk = getCurrentBlock();
10:    if vfd_hash.containsKey(blk.name) == null then
11:        /* call vRead_open() to get the vRead descriptor */
12:        vfd = vRead_open(blk.name, datanode_id);
13:        vfd_hash.put(blk.name, vfd);
14:    else
15:        vfd = vfd_hash.get(blk.name);
16:    end if
17:    /* read the data with vRead descriptor */
18:    if vfd != null then
19:        result = vRead_read(vfd, buf, off, len);
20:    else
21:        /* original method of HDFS */
22:        result = read_buffer(blk, buf, 0, len);
23:    end if
24:    if result > 0 then
25:        position += result;
26:        if position == blk.size then
27:            vRead_close(vfd);
28:        end if
29:    end if
30: end procedure
```

Algorithm 5 DFSInputStream read2 with vRead interface

```
1: vfd is the vRead descriptor for a given HDFS block
2: vfd_hash is the hashtable storing the mappings of HDFS block and vfd
3: datanode_id indicates the target datanode
4: blk is the instance of an HDFS block to read
5: position is the absolute start position of the target file stored in HDFS
6: buf is the application buffer
7: len is the number of bytes to read
8: off is the offset of the data block
9: procedure READ(position, buf, off, len)
10:     blk_list = getRangeBlock(position, len);
11:     remaining = len;
12:     for each blk in blk_list do
13:         start = position - blk.getStartOffset();
14:         bytesToRead = min(remaining, blk.size − start);
15:         if vfd_hash.containsKey(blk.name) == null then
16:             /* call vRead_open() to get the vRead descriptor */
17:             vfd = vRead_open(blk.name, datanode_id);
18:             vfd_hash.put(blk.name, vfd);
19:         else
20:             vfd = vfd_hash.get(blk.name);
21:         end if
22:         /* read the data with vRead descriptor */
23:         if vfd != null then
24:             result = vRead_read(vfd, buf, start, bytesToRead);
25:         else
26:             /* original method of HDFS */
27:             result = fetchBlocks(blk, start, bytesToRead, buf);
28:         end if
29:         remaining -= bytesToRead;
30:         position += bytesToRead;
31:     end for
32: end procedure
```

5 RELATED WORK

We have introduced some alternative solutions for optimizing I/O performance of virtualized cloud in the motivation sections of previous chapters. Here, we discuss other related work in the same area. These efforts can be divided into three categories: reducing device virtualization overhead, VM scheduling optimization, and functionality offloading.

5.1 Reducing Virtual Device Overhead

In recent years, many efforts have focused on reducing device virtualization overhead to improve VM I/O performance or capacity of VM hosts. vPipe [64] enables direct "piping" of application I/O data from source to sink devices, either files or TCP sockets, at the hypervisor level. By doing so, vPipe can avoid both device virtualization overhead and VM scheduling delays, resulting in better VM I/O performance. vPipe focuses on reducing the virtualization overhead between the virtual devices in the same VM, while vRead targets reducing the redundant data copies between VMs. Menon [65] proposes several optimizations such as offloading datagram checksum and TCP segmentation (TSO) to the Xen virtual machine monitor (VMM) [66] to improve TCP performance in Xen VMs. [67] aimed to reduce the TCP per-packet processing cost in VMs by packet coalescing to achieve better TCP receive performance. [68] proposes offloading part of the network device's functionality to the hypervisor to reduce CPU cycles consumed by network packet processing. These three work focus on optimizing some functionalities of TCP/IP in virtual environments, whereas vRead focuses on optimizing the data movement path between VMs communicating with each other and mainly targets the applications relying on HDFS.

Similarly, Ahmed *et al.* propose virtual interrupt coalescing for virtual SCSI controllers [69] to reduce disk I/O processing overhead in virtualized hosts. In [35, 70], Gordon *et al.* propose exit-less interrupt delivery mechanisms to mitigate the overhead of virtual

interrupt processing in KVM so that the incoming I/O events are sent to the destination VM without switching to the hypervisor by a VM-Exit. These two works can reduce the virtual interrupts' overhead incurred by processing disk or network I/O requests in a VM, but they cannot eliminate the unnecessary I/O flow between VMs which is targeted by vRead.

5.2 Scheduling Optimization

Since VM scheduling delay can significantly affect a VM's I/O performance in terms of throughput as well as application-perceived latency in virtual systems, many previous efforts have focused on reducing VM scheduling delay for I/O-intensive applications. [17] proposes a soft-realtime VM scheduler to reduce the response time of I/O requests thus improving the performance of soft-realtime applications such as media servers. However, its preemption-based policy may violate CPU fair-share if a VM is I/O-intensive. vSlicer [32] minimizes CPU scheduling delay and hence the application-perceived latency — to a certain degree by setting a smaller time-slice for latency-sensitive VMs. However, such a time-slice is not small enough to improve TCP/UDP throughput in LAN/datacenter environments. These two efforts both assume multiple VMs are running on the same CPU core. If there is no CPU sharing among VMs, they are less helpful. As new CPUs increasingly have more cores in each socket, the CPU sharing scenario is less common. vRead does not have any CPU sharing assumption, it also works no matter the VMs have dedicated cores or not. Besides, vRead reduces the I/O processing delay by avoiding redundant data copies between VMs thus eliminating the scheduling delay of I/O threads. MRG [71] proposes a VM scheduler specifically for Map/Reduce jobs. This scheduler keeps Map/Reduce job fairness by introducing a two-level group credit-based scheduling policy. The efficiency of map and reduce tasks can be improved by batching I/O requests within a group, hence superfluous context switches are eliminated. But, this work can not improve the I/O performance between Map/Reduce jobs and HDFS.

5.3 Functionality Offloading to the Hypervisor

Offloading partial I/O operations to reduce virtualization overhead and improve I/O performance is a well studied approach. [72] proposes the idea of offloading common middleware functionality to the hypervisor layer to reduce context switches between the guest OS and hypervisor. Differently, vRead introduces shortcutting at the inter-VM I/O level and is applicable to efficiently read files from other VMs' virtual disks. In [73], the whole TCP/IP stack is offloaded to a separate core to reduce the I/O response time of VMs sharing the same core. vSnoop [5] and vFlood [4] mitigated the negative impact of CPU access latency on TCP by offloading acknowledgement generation and congestion control to the driver domain of the Xen VMM. However, they all focus on the CPU sharing scenarios, but vRead is applicable no matter the VMs have dedicated cores or not. Besides, they are hardly applicable to inter-VM communication on the same host, which vRead addresses.

6 CONCLUSION

In order to reduce application running costs, both in terms of capital as well as operational expenditure, virtualization is widely used by most datacenters in the world so that multiple VMs can run in the same host simultaneously. However, directly moving appliations from traditional physical machines to the virtualized hosts may lead to sub-optimal I/O performance due to the server consolidation and device virtualization overhead. For example, as more VMs running in the same host, the CPU access delay experienced by each VM increases substantially. Meanwhile, the IRQ processing can also be delayed for the same reason. As a result, both I/O (network and disk) latency and throughput are significantly affected. Besides, compared with physical machine, all data movement within or among VMs suffers additional overhead incurred by the device virtualization resulting in higher CPU cycles consumption and lower I/O performance in return.

To solve these problems, in my disseratation, I proposed a series of mechanisms to mitigate the issues caused by virtualization. These methods include low-latency VM scheduler (vSlicer), I/O functionality offloading (vTurbo) and reducing device virtualization overhead for bigdata applications (vRead). For each approach, we implemented a prototype based on a popular hypervior and did comprehensive evaluation with it.

vSlicer supports a new class of CPU-sharing VMs called LSVMs. LSVMs improve the performance of I/O-bound applications by reducing the I/O processing latency; yet they do not violate the CPU share fairness among all VMs sharing the same CPU. vSlicer is based on the idea of differentiated-frequency CPU micro-slicing, where the regular time slice for an LSVM is further divided into smaller microslices for scheduling the LSVM multiple times within each scheduling round. Therefore, the LSVM is given more frequent accesses to the CPU for timely processing of I/O events. vSlicer is simple and generic for implementation in various hypervisors. Our evaluation of a Xen-based vSlicer prototype

demonstrates significant improvement at both network I/O and application levels over Xens credit scheduler.

vTurbo is a system that aims at accelerating I/O processing for VMs sharing the same core in a multi-core host. More specifically, vTurbo significantly reduces IRQ processing latency by dedicating one or more turbo core(s) to IRQ processing for all hosted VMs. The time-slice of a turbo core is magnitudes smaller than that of a regular core hence achieving negligible IRQ processing latency. vTurbo involves a VM scheduling policy that enforces fair sharing of both regular and turbo cores among VMs. Our evaluation of a vTurbo prototype shows that it vastly improves network and disk I/O throughput and consequently application-level performance for hosted VMs.

vRead is a system that directly improves the performance of HDFS. We observe that traditional virtual Hadoop systems frequently move data from a disk to a datanode VM which then sends the data to a client VM via the virtual network – regardless of if the two VMs are co-located or not. Thus, each HDFS read requires at least 5 data copies which incurs I/O overhead arising from device virtualization and CPU scheduling latency among VMs and I/O threads. vRead mitigates such penalty by shortcutting the HDFS reads at the hypervisor layer. Our evaluation of a vRead prototype shows that vRead can improve I/O throughput and reduce the CPU cost of HDFS. This benefits all applications (not limited to Hadoop) storing data in HDFS. Our application case studies demonstrate vReads applicability and effectiveness.

7 FUTURE WORK

7.1 Introduction

In chapter 4, we have introudced one efficient data access method for Hadoop in virtualized clouds named vRead [74] which enables direct ”reads” to the disk images of datanode VMs from the hypervisor, hence reducing device virtualization overhead and improving I/O throughput of HDFS. However, this method only benefits read I/O of HDFS deployed in virtual clusters. Clearly, many bigdata applications running in private or public cloud also perform write operation frequently such as uploading large file to HDFS or inserting records into Hadoop database (hbase). Compared with read, HDFS write is more complex. HDFS is designed to reliably store very large files across machines (VMs), so all blocks of a file are replicated for fault tolerance. For each HDFS write in virtualized cloud, the data has to be moved among 3 datanode (by default) and written into different datanode VM which incurs significant overhead caused by unnecessary data copies and device virtualization.

Our previous work vPipe [64] also generates a short cut on the data movement path avoiding unnecessary data moving from hypervisor to VM, hence reducing device virtualization overhead. This method work for both read and write operations. While, it has two limitations. First, it only works well for piped I/O in which data does not need to be modified by applications running in VMs. However, HDFS has to touch the data when moving data between network and disk to achieve some specific effects (e.g. maintaining data integrity). Second, vPipe targets the shortcuts between virtual devices within the same VM while multiple inter-VM data copies are involved for each HDFS write.

To benefit HDFS write in virtualized cloud, in the future, besides offloading some functionalities of applications to hypervisor to finish the data movement among devices without VMs involvement, we also try to offload data movement to storage subsystem with the help

of locality information of sink device and source device, which can further avoid unnecessary data copies and reduce the CPU consumption of VM hosts.

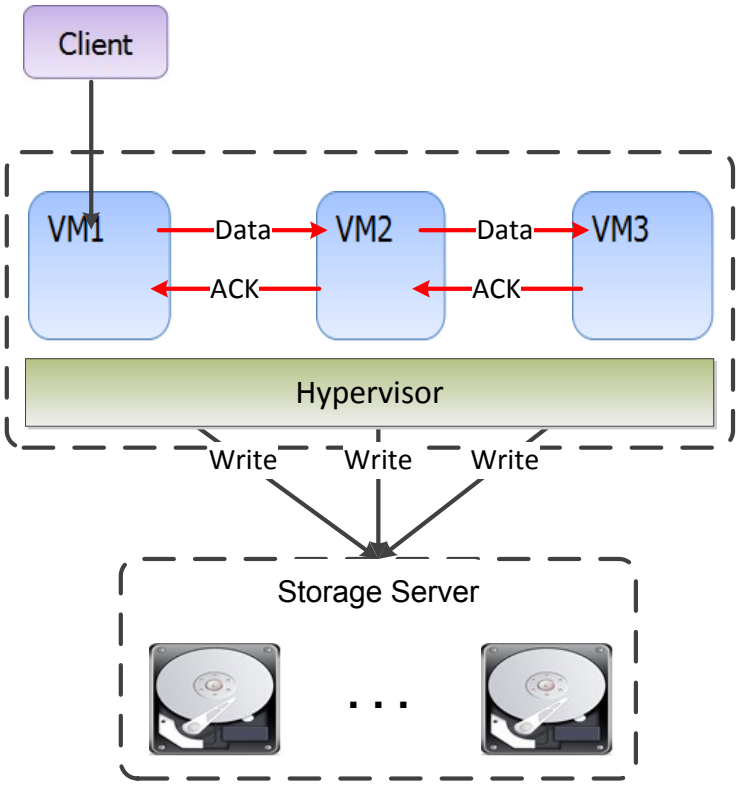


Figure 7.1.: HDFS write with default replica policy.

7.2 Design

Existing VM I/O optimization approaches ingore the locality of source and sink VMs, which may incur more overhead when moving data among VMs for bigdata applications. Lets still take the HDFS for example which is shown in Figure 7.1. In HDFS, all blocks of a file are replicated for better fault tolerance. When one file is written to HDFS, by default, three replications are written to three different datanode VM respectively. To minimize global bandwidth consumption and read latency, HDFS tries to satisfy a read request from a replica that is closest to the reader while still avoid single-point failure. Therefore,

the default replica placement is that: (If the host is virtualized) one replica is placed on a VM running in the same host as the client, the other two are placed on two VMs running in another physical machine. In virtualized cloud, physical hosts usually share the same storage to ease the VM live migration within the datacenter. So each replica datanode need to write the same data into the same storage during the HDFS write procedure, resulting in redundant data movement among 3 VMs and 3 disk writes for the same data. If this operation can cooperate with hypervisor, we can avoid the replica movement between datanode VMs and just write data back to disk once instead of three times. This can significantly save disk/network bandwidth and CPU cycles for data write of HDFS or other cloud file systems such as QFS and ClusterFS.

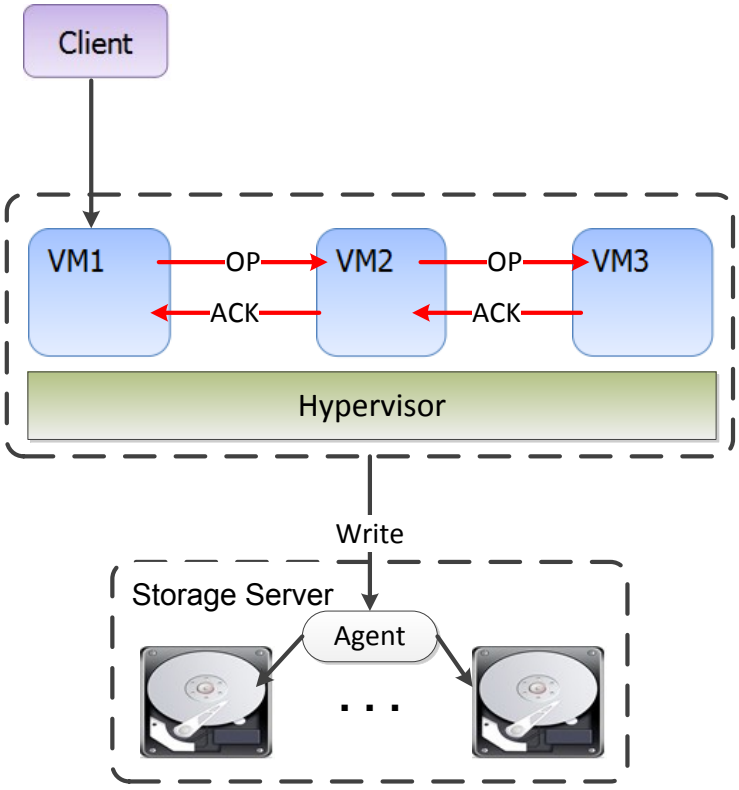


Figure 7.2.: HDFS write with shortcut policy.

To slove the problem discussed above, we can offload the data movement to hypervisor even storage subsystem which can further save CPU cycles of VM hosts. For instance, when we plan to build a virtual hadoop cluster in a virtualized data center. We usually just configure one VM node and copy the hadoop directory to other nodes via scp command. Even though the inter-VM communication in one host can be boosted by memory copy channel, it is still CPU consuming. Since all VMs running in the same datacenter usually share the same backend storage server, we can offload the data copy to storage server (done by the agent component) to save CPU cycles of the VM host.

After building the hadoop cluster, we need to upload file to HDFS then run MapReduce workload. This involves the HDFS write first. For each block of the uploaded file, there will be three replication written to three different HDFS datanode VMs respectively. If these VMs run in the same host or share the same storage server, we can avoid the data movement among VMs and only write data to storage server once instead of three times. This optimization is shown in Figure 7.2. For each received block, we write it back to storage server and buffer it there when the first VM tries to write the block to its local disk (virtul disk). We only send the identifier of the block instead of the real data among VMs with the help of functionality offloading to hypervisor. When we get the address information where other two VMs try to write, we let the storage server write the bufferred data to the actual address localy instead of sending the data to storage again. The whole process only incur one data write in the host and one data copy between host and storage server. This offloading immediately offers two advantages. First, it saves many CPU cycles to copy data among VMs and send data from VM host to storage server. Second, the data movement is faster than vanilla case because we only transfer the identifier of data block among VMs which is much cheaper than moving actual data.

REFERENCES

REFERENCES

[1] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[2] Microsoft Cloud Platform (Microsoft Azure). http://www.windowsazure.com/.

[3] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *USENIX OSDI*, 2002.

[4] Sahan Gamage, Ardalan Kangarlou, Ramana Rao Kompella, and Dongyan Xu. Opportunistic flooding to improve TCP transmit performance in virtualized clouds. In *ACM SoCC*, 2011.

[5] Ardalan Kangarlou, Sahan Gamage, Ramana Rao Kompella, and Dongyan Xu. vSnoop: Improving TCP throughput in virtualized environments via acknowledgement offload. In *ACM/IEEE SC*, 2010.

[6] Carl Waldspurger and Mendel Rosenblum. I/O virtulization. In *Communications of the ACM*, 2012.

[7] Mukil Kesavan, Ada Gavrilovska, and Karsten Schwan. Differential Virtual Time (DVT): Rethinking I/O service differentiation for virtual machines. In *ACM SoCC*, 2010.

[8] D. Patnaik, A.S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. Performance implications of hosting enterprise telephony applications on virtualized multi-core platforms. In *Principles, Systems and Applications of IP Telecommunications (IPTComm)*, 2009.

[9] Konstantin Shvachko, Sanjay Radia Hairong Kuang, and Robert Chansler. The Hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[10] Docker. http://www.docker.com/.

[11] Hadoop virtualization extensions on VMware vSphere5. In *VMware technical white paper*, 2012.

[12] A benchmarking case study of virtualized Hadoop performance on VMware vSphere5. In *VMware technical white paper*, 2013.

[13] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SOSP*, 2003.

[14] N. Nishiguchi. Evaluation and consideration of the credit scheduler for client virtualization. In *Xen Summit Asia 2008*, 2008.

[15] Sriram Govindan, Arjun R. Nath, Amitayu Das, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Xen and Co.: communication-aware CPU scheduling for consolidated Xen-based hosting platforms. In *ACM VEE*, 2007.

[16] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joowon Lee. Task-aware virtual machine scheduling for i/o performance. In *ACM VEE*, 2009.

[17] Min Lee, A. S. Krishnakumar, P. Krishnan, Navjot Singh, and Shalini Yajnik. Supporting soft real-time tasks in the Xen hypervisor. In *ACM VEE*, 2010.

[18] Yanyan Hu, Xiang Long, Jiong Zhang, Jun He, and Li Xia. I/O scheduling model of virtual machine based on multi-core dynamical partitioning. In *ACM HPDC*, 2010.

[19] Rackspace Cloud. http://www.rackspace.com.

[20] Gogrid Cloud. http://www.gogrid.com.

[21] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three CPU schedulers in Xen. *SIGMETRICS Performormance Evaluation Review*, 35(2):42–51, 2007.

[22] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *ACM SOSP*, 1999.

[23] J. McCalpin. The STREAM benchmark. http://www.cs.virginia.edu/stream/.

[24] VMware ESX. http://www.vmware.com/products/esx/.

[25] Lookbusy-a synthetic load generator. http://www.devin.com/lookbusy/.

[26] The Iperf Benchmark. http://www.noc.ucf.edu/Tools/Iperf/.

[27] Httperf. http://www.hpl.hp.com/research/linux/httperf/.

[28] Intel MPI benchmark. http://software.intel.com/en-us/articles/intel-mpi-benchmarks/.

[29] MPICH2. http://www.mcs.anl.gov/research/projects/mpich2/.

[30] Myconnection Server. http://www.myconnectionserver.com/.

[31] RFC 2326:Real Time Streaming Protocol (RTSP). http://rfc-ref.org/RFC-TEXTS/2326/chapter10.html.

[32] Cong Xu, Sahan Gamage, Pawan N. Rao, Ardalan Kangarlou, Ramana Rao Kompella, and Dongyan Xu. vSlicer: Latency-aware virtual machine scheduling via differentiated-frequency CPU slicing. In *ACM HPDC*, 2012.

[33] Luwei Cheng and Cho-Li Wang. vBalance: Using interrupt load balance to improve I/O performance for SMP virtual machines. In *ACM SoCC*, 2012.

[34] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving network connection locality on multicore systems. In *ACM EuroSys*, 2012.

[35] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: bare-metal performance for I/O virtualization. In *ACM ASPLOS*, 2012.

[36] Steen Larsen, Parthasarathy Sarangam, Ram Huggahalli, and Siddharth Kulkarni. Architectural breakdown of end-to-end latency in a TCP/IP network. *International Journal of Parallel Programming*, 37(6):556–571, 2009.

[37] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: predictable low latency for data center applications. In *ACM SoCC*, 2012.

[38] CPU isolation extensions. http://lwn.net/Articles/270623/.

[39] Apache Olio. http://http://incubator.apache.org/olio/.

[40] IOzone Filesystem Benchmark. http://www.iozone.org/.

[41] The Netperf Benchmark. http://www.netperf.org.

[42] Dhruba. Borthakur. The hadoop distributed file system: Architecture and design. In *Hadoop Project Website*, volume 11, page 21, 2007.

[43] Elastic Map/Reduce (EMR). http://aws.amazon.com/elasticmapreduce/.

[44] Omar Sefraoui and Mohsine Eleuldj Mohammed Aissaoui. Openstack: toward an open-source solution for cloud computing. In *International Journal of Computer Applications*, volume 55, 2012.

[45] Sahara. https://wiki.openstack.org/wiki/Sahara.

[46] Ro Recio, P. Culley, D. Garcia, J. Hilland, and B. Metzler. An overview of RDMA protocol specification. In *IETF Internet-draft draft-ietf-rddp-rdmap-03*, 2005.

[47] Hari Subramoni, Miao Luo Ping Lai, and Dhabaleswar K. Panda. RDMA over Ethernet – A preliminary study. In *In Cluster Computing and Workshops (CLUSTER)*, 2009.

[48] Rusty. Russell. Virtio – towards a de-facto standard for virtual I/O devices. In *ACM SIGOPS Operating Systems Review*, 2008.

[49] HDFS Short-Circuit Local Reads. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html.

[50] Venkateswararao Jujjuri, Eric Van Hensbergen, and Anthony Liguori. VirtFS – a virtualization aware file system pass-through. In *OLS*, 2010.

[51] Xiaolan Zhang, Pankaj Rohatgi Suzanne McIntosh, and John Linwood Griffin. XenSocket: A high-throughput interdomain transport for virtual machines. In *Middleware*, 2007.

[52] Macdonell, A. Wolfe Gordon Cam, Xiaodi Ke, and Paul Lu. Low-latency, high-bandwidth use cases for nahanni/ivshmem. In *KVM Forum*, 2011.

[53] Hamid Reza Mohebbi, Omid Kashefi, and Mohsen Sharifi. Zivm: A zero-copy inter-vm communication mechanism for cloud computing. *Computer and Information Science*, 4(6), 2011.

[54] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The quantcast file system. *Proceedings of the VLDB Endowment*, 6(11):1092–1101, 2013.

[55] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[56] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *Linux Symposim*, 2007.

[57] CPU Frequency Utils. http://mirrors.dotsrc.org/linux/utils/kernel/cpufreq/cpufrequtils.html.

[58] Apache HBase. http://hbase.apache.org/.

[59] Apache Hive. https://hive.apache.org/.

[60] Apache Sqoop. http://sqoop.apache.org/.

[61] Yaozu Dong, Zhao Yu, and Greg Rose. SR-IOV networking in Xen: architecture, design and implementation. In *WIOV*, 2008.

[62] R. Hiremane. Intel virtualization technology for directed I/O (Intel VT-d). *Technology@ Intel Magazine*, 4(10), 2007.

[63] Clark, Christopher, Steven Hand Keir Fraser, Eric Jul Jacob Gorm Hansen, Ian Pratt Christian Limpach, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation*, volume 2, pages 273–286, 2005.

[64] Sahan Gamage, Cong Xu, Ramana Rao Kompella, and Dongyan Xu. vPipe: Piped I/O offloading for efficient data movement in virtualized clouds. In *ACM SOCC*, 2014.

[65] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing network virtualization in Xen. In *USENIX ATC*, 2006.

[66] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SOSP*, 2003.

[67] Aravind Menon and Willy Zwaenepoel. Optimizing TCP receive performance. In *USENIX ATC*, 2008.

[68] Aravind Menon, Simon Schubert, and Willy Zwaenepoel. TwinDrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest OS drivers. In *ACM ASPLOS*, 2009.

[69] Irfan Ahmad, Ajay Gulati, and Ali Mashtizadeh. vIC: Interrupt coalescing for virtual machine storage device I/O. In *USENIX ATC*, 2011.

[70] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. Efficient and scalable paravirtual I/O system. In *USENIX ATC*, 2013.

[71] Hui Kang, Yao Chen, Jennifer L. Wong, Radu Sion, and Jason Wu. Enhancement of Xen's scheduler for MapReduce workloads. In *ACM HPDC*, 2011.

[72] Abel Gordon, Muli Ben-Yehuda, Dennis Filimonov, and Maor Dahan. VAMOS: virtualization aware middleware. In *WIOV*, 2011.

[73] Leah Shalev, Julian Satran, Eran Borovik, and Muli Ben-Yehuda. IsoStack: Highly efficient network processing on dedicated cores. In *USENIX ATC*, 2010.

[74] Cong Xu, Brendan Saltaformaggio, Sahan Gamage, Ramana Rao Kompella, and Dongyan Xu. vRead: Efficient data access for hadoop in virtualized clouds. In *ACM/IFIP/USENIX Middleware*, 2015.

VITA

VITA

Cong Xu earned his BS degree from Northeastern University in 2007 and MS degree from Beihang University in 2010. He obtained his PhD degree in computer science from Purdue University in 2015. His research interests focus on optimizing virtual machine I/O performance in the virtualized cloud. His research background spans the areas of virtualization technology, operating systems, distributed systems and system evaluation.